Tutorial 7: Developing a Simple Image Classifier

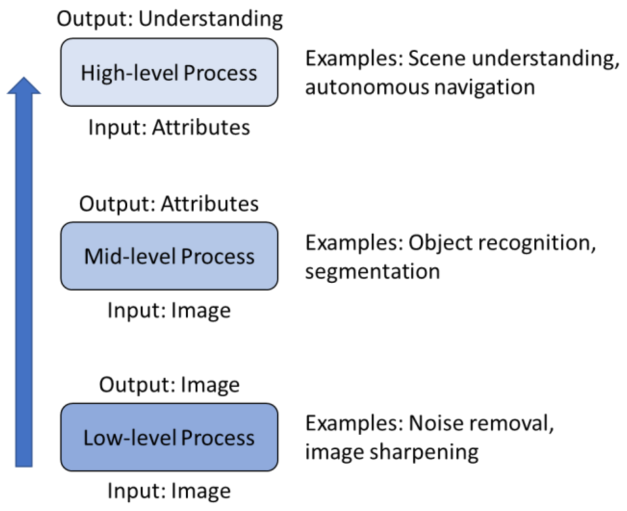
How to create a simple classifier in Matlab?

You already know from the Tutorial 5 the basic image processing techniques such as imread, imshow, image coordinates, RGB channels, binary/gray/color images, image thresholding, centroid, etc. Today, we will learn how to classify digits using a simple classifier. When you are navigating the robot, the robot should be able understand signs, shapes and objects. This tutorial covers a basic digit classifier which you can later extend for any low-level feature-based object classifier.

Low-level images features are edges, corners, blobs, colors and pixel intensities. They deal with the pixels directly. When we say low-level vision, that is more about the image processing.

Mid-level features are extracted from low-level features. They include attributes such as segmented images, motion of pixels, geometry of the scene, etc.

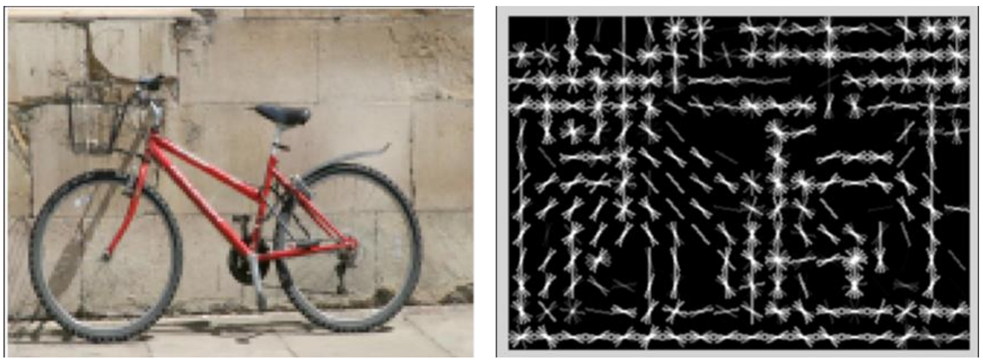
High-level algorithms are mostly in the machine learning domain. These algorithms are concerned with the interpretation or classification of a scene as a whole. Some examples are body pose classification, face detection, classification of human actions, object detection and recognition, etc. Based-on the complexity and the input/output, the image processing stages can be broadly categorised as follows.



In this tutorial, we will start with low-level features and perform low-level and mid-level image processing.

Histogram of Oriented Gradients (HOG) features

We use Histogram of Oriented Gradients (HOG) features as the feature descriptor of images. Below figure shows the an original image and extracted HOG features.



(Image source: http://6.869.csail.mit.edu/fa17/lecture/lecture19objectrecogntion1.pdf)

What is a Feature Descriptor?

A feature descriptor is a representation of an image or an image patch that simplifies the image by extracting useful information. Typically, a feature descriptor converts an image of size width x height x 3 (channels) to a feature vector / array of length n. Feature vector is very useful for tasks like image recognition and object detection.

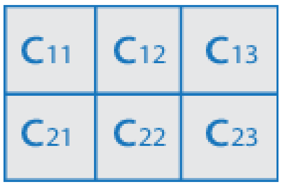
HOG is a feature descriptor used to detect objects in computer vision and image processing. The HOG descriptor technique counts occurrences of gradient orientation in localized portions of an image - detection window, or region of interest (ROI).

Implementation of the HOG descriptor algorithm is as follows:

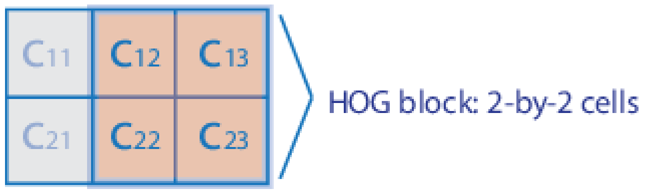
1. Divide the image into small connected regions called cells, and for each cell compute a histogram of gradient directions or edge orientations for the pixels within the cell.
2. Discretize each cell into angular bins according to the gradient orientation.
3. Each cell's pixel contributes weighted gradient to its corresponding angular bin.
4. Groups of adjacent cells are considered as spatial regions called blocks. The grouping of cells into a block is the basis for grouping and normalization of histograms.
5. Normalized group of histograms represents the block histogram. The set of these block histograms represents the descriptor.

The following illustrations demonstrate the algorithm implementation scheme (arrangement of Histograms in HOG Feature Vectors):

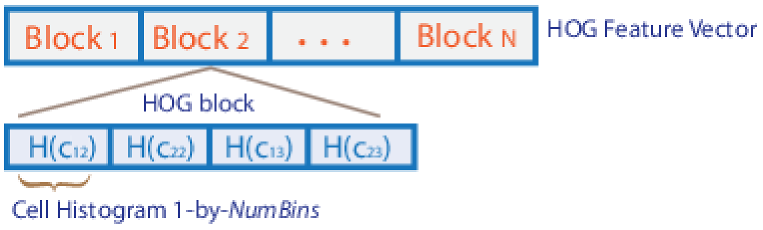
The figure below shows an image with six cells.



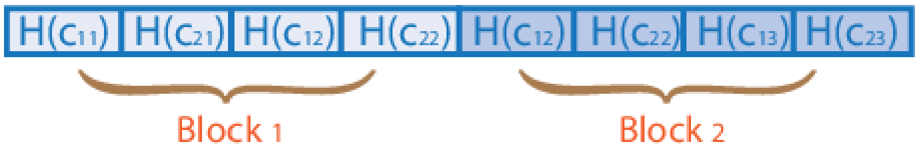
If you set the BlockSize to [2 2], it would make the size of each HOG block, 2-by-2 cells. The size of the cells are in pixels. You can set it with the CellSize property.



The HOG feature vector is arranged by HOG blocks. The cell histogram, $H(C_{yx})$, is 1-by-NumBins.



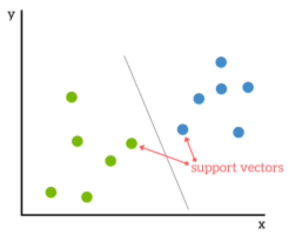
The figure below shows the HOG feature vector with a 1-by-1 cell overlap between blocks.



After extracting HOG features, we train a support vector machine (SVM) classifier. This trained classifier estimates the label of our input (test) image(s). These estimations are also called predictions. That means, when we provide the HOG features of test image to the classifier it predicts the class (also called the label) of that test image.

Support Vector Machines - What are they?

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be employed for both classification and regression purposes. SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown in the image below.



Support Vectors

Support vectors are the data points nearest to the hyperplane, the points of a data set that, if removed, would alter the position of the dividing hyperplane. Because of this, they can be considered the critical elements of a data set.

What is a hyperplane?

As a simple example, for a classification task with only two features (like the image above), you can think of a hyperplane as a line that linearly separates and classifies a set of data.
Intuitively, the further from the hyperplane our data points lie, the more confident we are that they have been correctly classified. We therefore want our data points to be as far away from the hyperplane as possible, while still being on the correct side of it.
So, when new testing data is added, whatever side of the hyperplane it lands will decide the class that we assign to it.

The above SVM introduction is taken from this web link:

- https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html

You can find good explanations of SVM from the below references:

- http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf
- http://www.robots.ox.ac.uk/~az/lectures/ml/lect2.pdf

Digit Classification Using HOG Features

Object classification is an important task in many computer vision applications, including surveillance, automotive safety, and image retrieval. For example, in an automotive safety application, you may need to classify nearby objects as pedestrians or vehicles. Regardless of the type of object being classified, the basic procedure for creating an object classifier is:

- Acquire a labelled data set with images of the desired object.
- Partition the data set into a training set and a test set.
- Train the classifier using features extracted from the training set.
- Test the classifier using features extracted from the test set.

To illustrate, this example shows how to classify numerical digits using HOG (Histogram of Oriented Gradient) features and a multiclass SVM (Support Vector Machine) classifier.

Digit Data Set

Synthetic digit images are used for training. The training images each contain a digit surrounded by other digits, which mimics how digits are normally seen together. Using synthetic images is convenient and it enables the creation of a variety of training samples without having to manually collect them. For testing, scans of handwritten digits are used to validate how well the classifier performs on data that is different than the training data. Although this is not the most representative data set, there is enough data to train and test a classifier, and show the feasibility of the approach.

Exercise 1

Create a new file and save it as 'Tute7_1.m'. Note: you need this (Exercise 2) code to complete your next practical. Run one code block, understand it and then go to next one.

```
clear all, close all

% Load training and test data using |imageDatastore|. Modify the path as
required.
syntheticDir   = fullfile('E:\code\Prac8\synthetic');
handwrittenDir = fullfile('E:\code\Prac8\handwritten');
```

```
% |imageDatastore| recursively scans the directory tree containing the
% images. Folder names are automatically used as labels for each image.
trainingSet = imageDatastore(syntheticDir,   'IncludeSubfolders', true,
'LabelSource', 'foldernames');
testSet     = imageDatastore(handwrittenDir, 'IncludeSubfolders', true,
'LabelSource', 'foldernames');
```

Use countEachLabel to tabulate the number of images associated with each label. In this example, the training set consists of 101 images for each of the 10 digits. The test set consists of 12 images per digit.

```
countEachLabel(trainingSet)

countEachLabel(testSet)
```

Show a few of the training and test images

```
figure;

subplot(2,3,1);
imshow(trainingSet.Files{102});

subplot(2,3,2);
imshow(trainingSet.Files{304});

subplot(2,3,3);
imshow(trainingSet.Files{809});

subplot(2,3,4);
imshow(testSet.Files{13});

subplot(2,3,5);
imshow(testSet.Files{37});

subplot(2,3,6);
imshow(testSet.Files{97});
```

Prior to training and testing a classifier, a pre-processing step is applied to remove noise artefacts introduced while collecting the image samples. This provides better feature vectors for training the classifier.

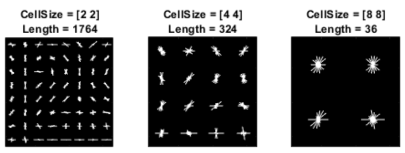
```
% Show pre-processing results
exTestImage = readimage(testSet,37);
processedImage = imbinarize(rgb2gray(exTestImage));
```

```
figure;

subplot(1,2,1)
imshow(exTestImage)

subplot(1,2,2)
imshow(processedImage)
```

Using HOG Features

The data used to train the classifier are HOG feature vectors extracted from the training images. Therefore, it is important to make sure the HOG feature vector encodes the right amount of information about the object. The extractHOGFeatures function returns a visualization output that can help form some intuition about just what the "right amount of information" means. By varying the HOG cell size parameter and visualizing the result, you can see the effect the cell size parameter has on the amount of shape information encoded in the feature vector:

```
img = readimage(trainingSet, 206);

% Extract HOG features and HOG visualization
[hog_2x2, vis2x2] = extractHOGFeatures(img,'CellSize',[2 2]);
[hog_4x4, vis4x4] = extractHOGFeatures(img,'CellSize',[4 4]);
[hog_8x8, vis8x8] = extractHOGFeatures(img,'CellSize',[8 8]);

% Show the original image
figure;
subplot(2,3,1:3); imshow(img);

% Visualize the HOG features
subplot(2,3,4);
plot(vis2x2);
title({'CellSize = [2 2]'; ['Length = ' num2str(length(hog_2x2))]});

subplot(2,3,5);
plot(vis4x4);
title({'CellSize = [4 4]'; ['Length = ' num2str(length(hog_4x4))]});

subplot(2,3,6);
plot(vis8x8);
title({'CellSize = [8 8]'; ['Length = ' num2str(length(hog_8x8))]});
```



The visualization shows that a cell size of [8 8] does not encode much shape information, while a cell size of [2 2] encodes a lot of shape information but increases the dimensionality of the HOG feature vector significantly. A good compromise is a 4-by-4 cell size. This size setting encodes enough spatial information to visually identify a digit shape while limiting the number of dimensions in the HOG feature vector, which helps speed up training. In practice, the HOG parameters should be varied with repeated classifier training and testing to identify the optimal parameter settings.

```
cellSize = [4 4];
hogFeatureSize = length(hog_4x4);
```

Train a Digit Classifier

Digit classification is a multiclass classification problem, where you have to classify an image into one out of the ten possible digit classes. In this example, the fitcecoc function from the Statistics and Machine Learning Toolbox is used to create a multiclass classifier using binary SVMs.

Start by extracting HOG features from the training set. These features will be used to train the classifier.

```
% Loop over the trainingSet and extract HOG features from each image. A
% similar procedure will be used to extract features from the testSet.

numImages = numel(trainingSet.Files);
trainingFeatures = zeros(numImages, 324, 'single');

for i = 1:numImages
    img = readimage(trainingSet, i);

    img = rgb2gray(img);

    % Apply pre-processing steps
    img = imbinarize(img);

    trainingFeatures(i, :) = extractHOGFeatures(img, 'CellSize', [4 4]);
end

% Get labels for each image.
trainingLabels = trainingSet.Labels;
```

Next, train a classifier using the extracted features.

```
% fitcecoc uses SVM learners and a 'One-vs-One' encoding scheme.
classifier = fitcecoc(trainingFeatures, trainingLabels);
```

Evaluate the Digit Classifier

Evaluate the digit classifier using images from the test set, and generate a confusion matrix to quantify the classifier accuracy.

As in the training step, first extract HOG features from the test images. These features will be used to make predictions using the trained classifier.

```
% Extract HOG features from the test set. The procedure is similar to what
% was shown earlier and is encapsulated as a helper function for brevity.
[testFeatures, testLabels] = helperExtractHOGFeaturesFromImageSet(testSet,
hogFeatureSize, cellSize);

% Make class predictions using the test features.
predictedLabels = predict(classifier, testFeatures);

% Tabulate the results using a confusion matrix.
confMat = confusionmat(testLabels, predictedLabels);

helperDisplayConfusionMatrix(confMat)
```

```
digit  | 0      1      2      3      4      5      6      7      8      9
--------------------------------------------------------------------------------
0      | 0.25   0.00   0.08   0.00   0.00   0.00   0.58   0.00   0.08   0.00
1      | 0.00   0.75   0.00   0.00   0.08   0.00   0.00   0.08   0.08   0.00
2      | 0.00   0.00   0.67   0.17   0.00   0.00   0.08   0.00   0.00   0.08
3      | 0.00   0.00   0.00   0.58   0.00   0.00   0.33   0.00   0.00   0.08
4      | 0.00   0.08   0.00   0.17   0.75   0.00   0.00   0.00   0.00   0.00
5      | 0.00   0.00   0.00   0.00   0.00   0.33   0.58   0.00   0.08   0.00
6      | 0.00   0.00   0.00   0.00   0.25   0.00   0.67   0.00   0.08   0.00
7      | 0.00   0.08   0.08   0.33   0.00   0.00   0.17   0.25   0.00   0.08
8      | 0.00   0.00   0.00   0.08   0.00   0.00   0.00   0.08   0.67   0.17
9      | 0.00   0.08   0.00   0.25   0.17   0.00   0.08   0.00   0.00   0.42
```

The table shows the confusion matrix in percentage form. The columns of the matrix represent the predicted labels, while the rows represent the known labels. For this test set, digit 0 is often misclassified as 6, most likely due to their similar shapes. Similar errors are seen for 9 and 3. Training with a more representative data set like MNIST or SVHN, which contain thousands of handwritten characters, is likely to produce a better classifier compared with the one created using this synthetic data set.

Summary

This example illustrated the basic procedure for creating a multiclass object classifier using the extractHOGfeatures function from the Computer Vision Toolbox and the fitcecoc function from the Statistics and Machine Learning Toolbox. Although HOG features and an ECOC classifier were used here, other features and machine learning algorithms can be used in the same way. For instance, you can explore using different feature types for training the classifier; or you can see the effect of using other machine learning algorithms available in the Statistics and Machine Learning Toolbox such as k-nearest neighbours.

Supporting Functions

```
function helperDisplayConfusionMatrix(confMat)
% Display the confusion matrix in a formatted table.

% Convert confusion matrix into percentage form
confMat = bsxfun(@rdivide,confMat,sum(confMat,2));

digits = '0':'9';
colHeadings = arrayfun(@(x)sprintf('%d',x),0:9,'UniformOutput',false);
format = repmat('%-9s',1,11);
header = sprintf(format,'digit  |',colHeadings{:});
fprintf('\n%s\n%s\n',header,repmat('-',size(header)));
for idx = 1:numel(digits)
    fprintf('%-9s',   [digits(idx) '      |']);
    fprintf('%-9.2f', confMat(idx,:));
    fprintf('\n')
end
end

function [features, setLabels] = helperExtractHOGFeaturesFromImageSet(imds,
hogFeatureSize, cellSize)
% Extract HOG features from an imageDatastore.

setLabels = imds.Labels;
numImages = numel(imds.Files);
features  = zeros(numImages, hogFeatureSize, 'single');

% Process each image and extract features
for j = 1:numImages
    img = readimage(imds, j);
    img = rgb2gray(img);

    % Apply pre-processing steps
    img = imbinarize(img);

    features(j, :) = extractHOGFeatures(img,'CellSize',cellSize);
end
end
```

Reference: https://au.mathworks.com/help/vision/examples/digit-classification-using-hog-features.html

Exercise 2

Let's modify the above code to classify only four classes (digit 1 to 4). You are given four images. You need to classify only one image at a time.

- Make a copy of 'Tute7_1.m' and rename it as 'Tute7_2.m'.
- Delete the codes correspond to figures.
- Delete the last two lines correspond to confusion matrix.

- Note from the workspace that the following value gets assigned for `hogFeatureSize`:
  ```
  hogFeatureSize = 324;
  ```

- We can hardcode that value with above line by replacing the old line
  ```
  (hogFeatureSize = length(hog_4x4);)
  ```

- Before moving to the next step, run the code and see if everything works.
- Now, we have the same code but only with the core functionalities.
- This code uses a test set. Let's change the test set to a single test image.
- You can delete the below three lines correspond to testSet.

```
handwrittenDir = fullfile('E:\code\Prac8\handwritten');

testSet     = imageDatastore(handwrittenDir, 'IncludeSubfolders', true,
'LabelSource', 'foldernames');

countEachLabel(testSet)
```

- Below line is used to extract HOG features from a test dataset arranged in folders.

```
[testFeatures, testLabels] = helperExtractHOGFeaturesFromImageSet(testSet,
hogFeatureSize, cellSize);
```

- We need to replace above line with the below line.

```
[testFeatures,visualization] = extractHOGFeatures(I2,'CellSize',cellSize);
```

- Here, I2 is the input image. Before extracting the HOG features, input image should be resized to match the size of training images. You can read and resize the image as follows. You need to find **X Y** values and enter them in the code.

```
I1 = imread('E:\code\Prac8\3.jpg');
figure,imshow(I1)
I2 = imresize(I1, [X Y]);
```

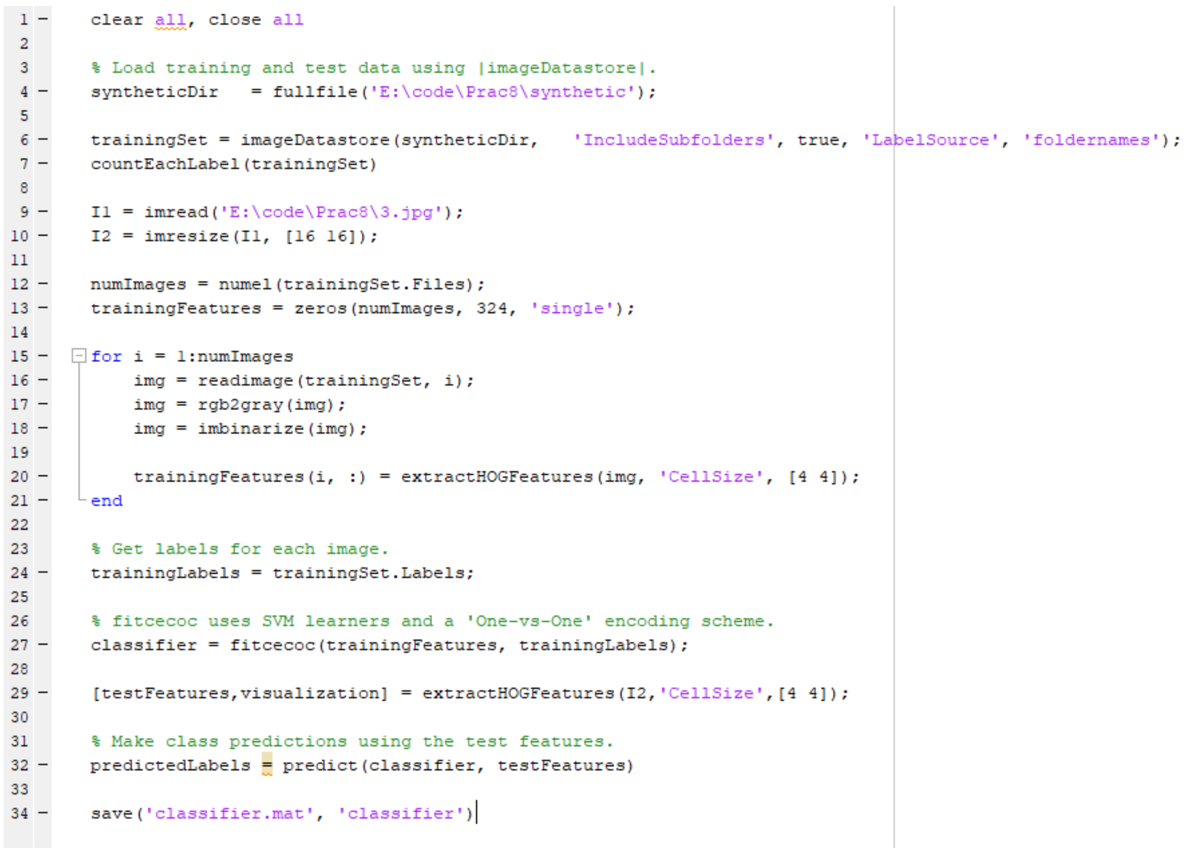
- Now, your code is complete. When you run it you should see the correct test label under 'predictedLabels'.
- Since we are interested in only four classes, you can delete the unwanted class folders inside 'synthetic' directory.
- You can delete the two functions.
- Delete below two lines and enter the value for 'cellSize' manually where necessary.

```
cellSize = [4 4];
hogFeatureSize = 324;
```

- Add this line to your code, so you can use the trained classifier with your other examples.

```
save('classifier.mat', 'classifier')
```

- Your final code should look like below screenshot.



```
clear all, close all

% Load training and test data using |imageDatastore|.
syntheticDir   = fullfile('E:\code\Prac8\synthetic');

trainingSet = imageDatastore(syntheticDir,   'IncludeSubfolders', true, 'LabelSource', 'foldernames');
countEachLabel(trainingSet)

I1 = imread('E:\code\Prac8\3.jpg');
I2 = imresize(I1, [16 16]);

numImages = numel(trainingSet.Files);
trainingFeatures = zeros(numImages, 324, 'single');

for i = 1:numImages
    img = readimage(trainingSet, i);
    img = rgb2gray(img);
    img = imbinarize(img);

    trainingFeatures(i, :) = extractHOGFeatures(img, 'CellSize', [4 4]);
end

% Get labels for each image.
trainingLabels = trainingSet.Labels;

% fitcecoc uses SVM learners and a 'One-vs-One' encoding scheme.
classifier = fitcecoc(trainingFeatures, trainingLabels);

[testFeatures,visualization] = extractHOGFeatures(I2,'CellSize',[4 4]);

% Make class predictions using the test features.
predictedLabels = predict(classifier, testFeatures)

save('classifier.mat', 'classifier')
```

Exercise 3

Let’s simplify this code further.

- Make a copy of ‘Tute7_2.m’ and rename it as ‘Tute7_3.m’.
- You can delete the codes correspond to training and load the previously saved classifier.
- This is your final digit classification code.

```
clear all, close all

load ('classifier.mat');

I1 = imread('E:\code\Prac8\2.jpg');
I2 = imresize(I1, [16 16]);

[testFeatures,visualization] = extractHOGFeatures(I2,'CellSize',[4 4]);

% Make class predictions using the test features.
predictedLabels = predict(classifier, testFeatures)
```