Introduction to Machine Learning, Fall 2014 - Exercise session III

Rodion "rodde" Efremov
013593012

November 11, 2014

Problem 1 (3 points)

Problem 2 (3 points)

Problem 3 (3 points)

Problem 4 (15 points)

In this exercise we implement an extremely simple prototype-based classifier to classify handwritten digits from the MNIST dataset, and compare that to a nearest-neighbor classifier.

The included `ex3.m` file contains well document version of the code following. Also, if the `cd` command in the file is corrected to your path to MNIST data, it can be run with the `run('ex3.m')` command.

(a) Download the MNIST data from the course web page. In addition to the actual data, the package contains some functions for easily loading the data into Matlab/Octave/R and for displaying digits. See the README files for details. Load the first $N = 5,000$ images using the provided function.

First we need to get to the directory containing the `loadmnist.m` file, run it, and load 5000 images along their class labels.

```
cd /path/to/mnist
run('loadmnist.m');
[X y] = loadmnist(5000);
```

Now the matrix X contains an image at each row (`X(i,:)` is the ith image).

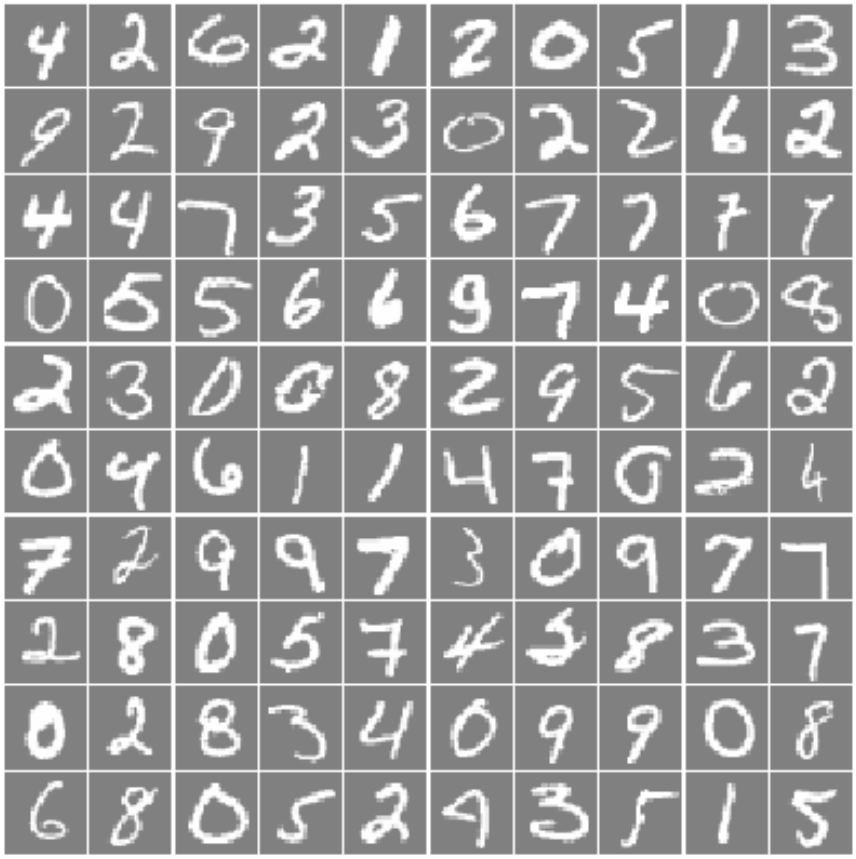
(b) Use the provided functions to plot a random sample of 100 handwritten digits, and show the associated labels. Verify that the labels match the digit images. (This is a sanity check that you have the data [is] in the right format.)

As to verify the data, we need a randomly selected array of indices:

```
indices = randperm(5000, 100);
```

After that we can draw the digits from the actual data and compare them to the actual labels:

```
visual(X(indices,:));
y(indices) # Prints the actual labels.
           # Appeared to be in accord with the drawn digits.
```



The first ten labels in vector `y` correspond to the digits in the upmost row: 4, 2, 6, 2, 1, 2, 0, 5, 1, 3, as expected.

(c) Divide the data into two parts: A 'training set' consisting of the first 2,500 images (and associated labels), and a 'test set' containing the remaining 2,500 images (and their associated labels).

Dividing the data set is simple:

```
TrainingSet = X(1:2500,:);
TestSet = X(2501:5000,:);
TrainingSetY = y(1:2500);
TestSetY = y(2501:5000);
```

(d) For each ot the ten classes (digits 0-9), compute a class *prototype* given by the mean of all the images in the training set that belong to this class. That is, select from the training set all images of class '0' and compute the mean image of these; this should look sort of like a zero. Do this for all ten classes, and plot the resulting images. Do they look like what you would expect?

First of all, we need a function to compute the prototypes:

```
function mean_vector = compute_prototype(T, labels, digit)
  mean_vector = zeros(1, size(T)(2));
  count = 0;
  for i = 1:size(T)(1)
    if labels(i) == digit
      mean_vector += T(i,:);
      count++;
    endif
  endfor
  mean_vector /= count;
endfunction
```

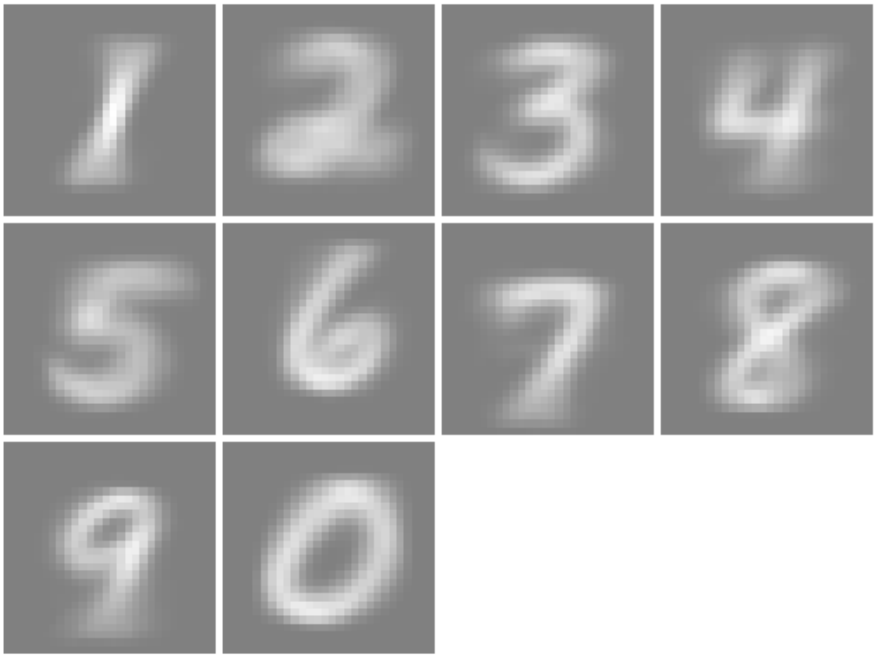
Next, let us construct a 10×784-matrix M, whose rows are the prototypes. For all digits $d \in \{1, 2, \ldots, 9\}$, the prototype for d is M's dth row, and for the digit 0, the prototype is M's 10th row.

```
function prototype_matrix = get_prototype_matrix(T, labels)
  prototype_matrix = zeros(10, 784);
  for i = 1:9
    prototype_matrix(i,:) = compute_prototype(T, labels, i);
  endfor
  prototype_matrix(10,:) = compute_prototype(T, labels, 0);
endfunction
```

Now, running

```
prototype_matrix = get_prototype_matrix(TrainingSet, TrainingSetY);
visual(prototype_matrix);
```

we will obtain the following output:



The prototypes look pretty good, but I suspect that 6 and 8 will be confused pretty often.

(e) For each of the images in the test set, compute the Euclidian distance of the image to all 10 prototypes, and classify the test image into the class for which the distance to the prototype is the smallest. So, if a test image is closer to the prototype for '3' than it is to the prototypes for any of the other digits, predict its class to be '3'. Compute and display the resulting *confusion matrix*.

Let us implement the naïve classifier:

```
function dist = euclidean_distance(a, b)
  dist = norm(a - b);
endfunction

function label = naive_classify_impl(image, prototype_matrix)
  best_digit = -1;
  best_dist = Inf;
  for digit = 1:10
    current_dist = euclidean_distance(image, prototype_matrix(digit,:));
```

```
    if best_dist > current_dist
      best_dist = current_dist;
      best_digit = digit;
    endif
  endfor
  if best_digit == 10
    label = 0;
  else
    label = best_digit;
  endif
endfunction

function cm = naive_classify(TestSet, TestSetY, prototype_matrix)
  cm = zeros(10, 10);
  for i = 1:size(TestSet)(1)
    predicted_class = naive_classify_impl(TestSet(i,:), prototype_matrix);
    actual_class = TestSetY(i);
    predicted_class_index = predicted_class;
    actual_class_index = actual_class;
    if predicted_class == 0
      predicted_class_index = 10;
    endif
    if actual_class == 0
      actual_class_index = 10;
    endif
    cm(predicted_class_index, actual_class_index)++;
  endfor
endfunction
```

Now, to test the classifier with the test data, and to print the resulting confusion matrix, all we need is

```
cm1 = naive_classify(TrainingSet,
                     TrainingSetY,
                     prototype_matrix)
```

And the confusion matrix looks like

```
   281    20     9     4    16    12    14    13     4     0
     1   192     4     1     1    15     1     9     6     1
     0     4   193     0    14     0     0    25     2     2
     0     5     2   202    11     9    15     4    38     0
     3     2    20     0   136     8     0    20     2     4
     0     2     0     5     2   198     0     1     1     6
     0     2     5     2     6     0   232     1    14     2
     1    11    11     0     1     0     1   150     2     2
```

```
   0     1     8    40    11     0    11    15   179     0
   0     2     1     1     9     5     1     2     3   228
```

where the rightmost column and the bottom row corresponds to digit '0'. The corresponding error rate is 0.2036.

(f) Classify each of the test images with a nearest neighbor classifier: For each of the test images, compute its Euclidian distance to all (2,500) of the training images, and let the predicted class be the class of the closest training image. Compute and display the resulting confusion matrix.

The confusion matrix is

```
 283     6     0     3     3     1     3     5     2     0
   0   216     3     0     0     0     2     9     0     0
   0     5   221     0     6     0     0    10     2     2
   1     0     2   231     0     1     6     1    12     0
   1     1    11     0   183     1     0     8     1     1
   0     1     2     0     5   240     0     0     2     1
   1     5     4     2     1     0   257     2    11     0
   0     5     5     0     1     0     0   196     0     0
   0     0     5    19     3     1     7     5   219     0
   0     2     0     0     5     3     0     4     2   241
```

And the implementation is:

```
function predicted_label = knn_classify_impl(TrainingSet,
                                             TrainingSetY,
                                             image)
    best_distance = Inf;
    best_label = -1;
    for i = 1:size(TrainingSet)(1)
      current_dist = euclidean_distance(image, TrainingSet(i,:));
      if best_distance > current_dist
        best_distance = current_dist;
        best_label = TrainingSetY(i);
      endif
    endfor
    predicted_label = best_label;
endfunction

function cm = knn_classify(TrainingSet, TrainingSetY, TestSet, TestSetY)
  cm = zeros(10, 10);
  for i = 1:size(TestSet)(1)
    predicted_digit = knn_classify_impl(TrainingSet,
                                        TrainingSetY,
                                        TestSet(i,:));
```

```
    % Map digits 0 to index 10.
    if predicted_digit == 0
      predicted_digit = 10;
    endif
    actual_digit = TestSetY(i);
    if actual_digit == 0
      actual_digit = 10;
    endif
    cm(predicted_digit, actual_digit)++;
  endfor
endfunction

function err_rate = compute_error_rate(cm)
  total = sum(cm(:));
  hits = 0;
  for i = 1:10
    hits += cm(i,i);
  endfor
  err_rate = 1 - hits / total;
endfunction
```

(g) Compute and compare the error rates of both classifiers (the prototype-based classifier and the nearest neighbor classifier). Which is working better? Based on the confusion matrix, which digits are confused with each other? Why do you think this is?

The error rate of the prototype-based classifier is 0.2036 and the error rate of kNN-classifier is 0.0852. I think this is due to the fact that kNN works in more global fasion as compared to the prototype-based one, yet the latter provides much better performance.