Programming Exercise 4: Multi-class Classification

Introduction

In this exercise, you will implement one-vs-all logistic regression to recognize hand-written digits. To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the cd command in Octave/MATLAB to change to this directory before starting this exercise.

Files included in this exercise

`Ex4.m` - Octave/MATLAB script that steps you through the exercise
`Ex4data1.mat` – Training dataset of hand-written digits
`displayData.m` - Function to help visualize the dataset
`fmincg.m` - Function minimization routine (similar to fminunc)
`sigmoid.m` – Sigmoid function
[*] `lrCostFunction.m` – Logistic regression cost function
[*] `oneVsAll.m` – Train a one-vs-all multi-class classifier
[*] `predictOneVsAll.m` – Predict using a one-vs-all multi-class classifier

For this exercise, you will use logistic regression to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classification task.

In this exercise, you will extend your previous implementation of logistic regression and apply it to one-vs-all classification.

Throughout the exercise, you will be using the script `ex4.m`. This script sets up the dataset for the problem and makes calls to functions that you will write. You do not need to modify this script. You are only required to modify functions in other files, by following the instructions in this assignment.

[*] indicates files you will need to complete

1 Dataset

You are given a data set in `ex4data1.mat` that contains 5000 training examples of handwritten digits[1]. The `.mat` format means that that the data has been saved in a native Octave/MATLAB matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the `load` command. After loading, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('ex4data1.mat');
% The matrices X and y will now be in your Octave environment
```

There are 5000 training examples in `ex4data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is “unrolled” into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix X. This gives us a 5000 by 400 matrix X where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} - - x^{(1)} - - \\ - - x^{(2)} - - \\ \vdots \\ - - x^{(m)} - - \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. To make things more compatible with Octave/MATLAB indexing, where there is no zero index, the digit zero is mapped to the value ten. Therefore, a “0” digit is labeled as “10”, while the digits “1” to “9” are labeled as “1” to “9” in their natural order.

[1] This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/)

2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of `ex4.m`, the code randomly selects 100 rows from X, calls them `test_data`, and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. The `displayData` function is provided, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.

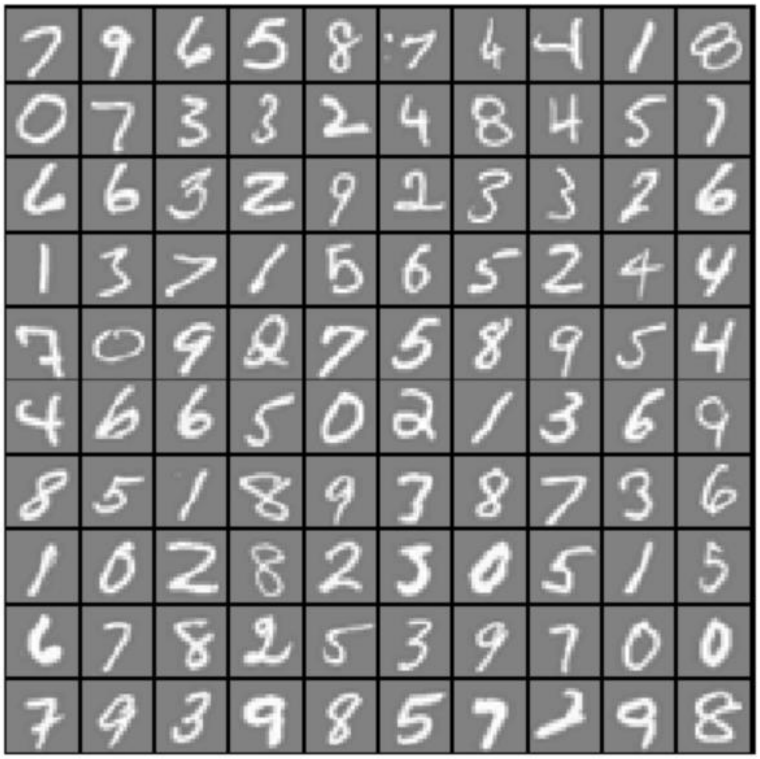

Figure 1: Examples from the dataset

3 Vectorizing logistic regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any for loops. You can use your code in the last exercise as a starting point for this exercise.

3.1 Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in logistic regression, the cost function is

$$J(\theta) = \sum_{i=1}^{m} [-y^{(i)} \log\left(h\left(x^{i}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \mathrm{h}\left(x^{(i)}\right)\right)].$$

To compute each element in the summation, we have to compute $h_\theta(x^{(i)})$ for every example i, where $h_\theta\left(x^{(i)}\right) = g\left(x^{(i)} * \theta\right)$ and $g(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. It turns out that we can compute this quickly for all our examples by using matrix multiplication. Let us define X and θ as:

$$X = \begin{bmatrix} --x^{(1)}-- \\ --x^{(2)}-- \\ \vdots \\ --x^{(m)}-- \end{bmatrix} \qquad \text{and} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}.$$

Then, by computing the matrix product $X * \theta$, we have

$$X * \theta = \begin{bmatrix} --x^{(1)} * \theta -- \\ --x^{(2)} * \theta -- \\ \vdots \\ --x^{(m)} * \theta -- \end{bmatrix}$$

This allows us to compute the products $x^{(i)} * \theta$ for all our examples i in one line of code.
Your job is to write the cost function in the file `lrCostFunction.m`. Your implementation should use the strategy presented above to calculate $x^{(i)} * \theta$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of `lrCostFunction.m` should not contain any loops. (Hint: You might want to use the element-wise multiplication operation (.*) and the sum operation `sum` when writing this function).

3.2 Vectorizing the gradient

Recall that the gradient of the logistic regression cost is a vector where the j^{th} element is defined as

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_{i=1}^{m} \left(h\left(x^{(i)}\right) - y^{(i)}\right)\left(x_j^{(i)}\right).$$

To vectorized this equation over the dataset, we start by writing out all the partial derivatives explicitly for all θ_j,

$$
\begin{aligned}
\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} &= \begin{bmatrix} \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \right) \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)} \right) \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)} \right) \\ \vdots \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)}) x_n^{(i)} \right) \end{bmatrix} \\
&= \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) x^{(i)} \right) \\
&= X^T (h_\theta(x) - y). \qquad (1)
\end{aligned}
$$

Where,

$$
h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}.
$$

Note that $x^{(i)}$ is a vector, while $h_\theta(x^{(i)}) - y^{(i)}$ is a scalar (single number). To understand the last step of the derivation, let $\beta_i = h_\theta(x^{(i)}) - y^{(i)}$ and observe that:

$$
\sum_i \beta_i x^{(i)} = \begin{bmatrix} | & | & & | \\ (x^{(1)})^T & (x^{(2)})^T & \dots & (x^{(m)})^T \\ | & | & & | \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T \beta,
$$

where the values $\beta_i = h_\theta(x^{(i)}) - y^{(i)}$.
The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, try to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations. You should now implement Equation 1 to compute the correct vectorized gradient. Once you are done, complete the function `lrCostFunction.m` by implementing the gradient.

Debugging Tip: Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the `size` function. For example, given a data matrix X of size 100 × 20 (100 examples, 20 features) and θ, a vector with dimensions 20 × 1, you can observe that $X * \theta$ is a valid multiplication operation, while $\theta * X$ is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

4 One-vs-all classification

In this part of the exercise, you will implement one-vs-all classification by training multiple logistic regression classifiers, one for each of the K classes in the dataset (Figure 1). In the handwritten digits dataset, K = 10, but your code should work for any value of K.

You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\theta \in \mathbb{R}^{(N+1)\times k}$, where each column of θ corresponds to the learned logistic regression parameters for one class. You can do this with a “for”-loop from 1 to K, training each classifier independently.

Note that the y argument to this function is a vector of labels from 1 to 10, where the digit “0” has been mapped to the label 10 (to avoid confusions with indexing).

When training the classifier for class $k \in \{1, 2, \ldots, K\}$, you will want a m-dimensional vector of labels y, where $y_j \in 0, 1$ indicates whether the j^{th} training instance belongs to class k $(y_j = 1)$, or if it belongs to a different class $(y_j = 0)$. You may find logical arrays helpful for this task.

Octave/MATLAB Tip: Logical arrays in Octave/MATLAB are arrays which contain binary (0 or 1) elements. In Octave/MATLAB, evaluating the expression `a == b` for a vector `a` (of size m × 1) and scalar `b` will return a vector of the same size as `a` with ones at positions where the elements of `a` are equal to `b` and zeroes where they are different. To see how this works for yourself, try the following code in Octave/MATLAB:

```
a = 1:10;      % Create a and b
b = 3;
a == b;        % You should try different values of b here
```

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more efficient for dealing with a large number of parameters.

After you have correctly completed the code for `oneVsAll.m`, the script `ex4.m` will continue to use your `oneVsAll` function to train a multi-class classifier.

4.1 One-vs-all prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the “probability” that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label (1, 2,..., or K) as the prediction for the input example.

You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions.

Once you are done, `ex4.m` will call your `predictOneVsAll` function using the learned values of θ. You should see that the accuracy is about 90% on the testing dataset (i.e., it classifies 90% of the examples in the testing set correctly).