Machine Learning Programming Exercise 2: Logistic Regression

adapted to Python language from Coursera/Andrew Ng

October 2, 2023

1 Introduction

In this exercise, you will implement logistic regression and apply it to two different datasets. Before starting on the programming exercise, we strongly recommend watching the video lectures. 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.

2 Files included in this exercise

- **ex2.py** - Python script that will help step you through the exercise
- **ex2__reg.py** - Python script for the second part of the exercise
- **ex2__multiclass.py** - Python script for the later parts of the exercise
- **ex2data1.txt** - Dataset for linear regression with one variable
- **ex2data2.txt** - Dataset for linear regression with multiple variables
- **ex2data3.txt** - Dataset for linear regression with multiple variables and multiple classes
- **plotDecisionBoundary.txt** - Plots the data points X and y into a new figure with the decision boundary defined by theta

⋆ **plotData.py** - function to display the dataset

⋆ **sigmoid.py** -Sigmoid Function

⋆ **gradientDescent.py** - Function to run gradient descent

⋆ **costFunction.py** - Logistic Regression Cost Function

⋆ **gradientFunction.py** - Logistic Regression Gradient Function

⋆ **costFunctionReg.py** - Regularized Logistic Regression Cost variables

⋆ **gradientFunctionReg.py** - Regularized Logistic Regression Gradient Function

⋆ **predict.py** - Logistic Regression Prediction Function

⋆ **lrCostFunction.py** - Logistic regression cost function

⋆ **lrCostGradient.py** - Logistic regression cost gradient

⋆ **learnOneVsAll.py** - Train a one-vs-all multi-class classifier

⋆ **predictOneVsAll.py** - Predict using a one-vs-all multi-class classifier

⋆ **predict.py** - Neural network prediction function

[⋆] indicates files you will need to complete

Throughout the exercise, you will be using the scripts **ex2.py**, **ex2__reg.py** and **ex2__multiclass.py**. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

3 Logistic Regression

You can start by watching this video!

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university. Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams. You have historical data from previous applicants that you can use as a training set for logistic regression. For each training example, you have the applicant's scores on two exams and the admissions decision. Your task is to build a classification model that estimates an applicant's probability of admission based on the scores from those two exams. This outline and the framework code in **ex2.py** will guide you through the exercise.

3.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of **ex2.py**, the code will load the data and display it on a 2-dimensional plot by calling the function **plotData**. You will now complete the code in **plotData** so that it displays a figure like Figure 1, where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

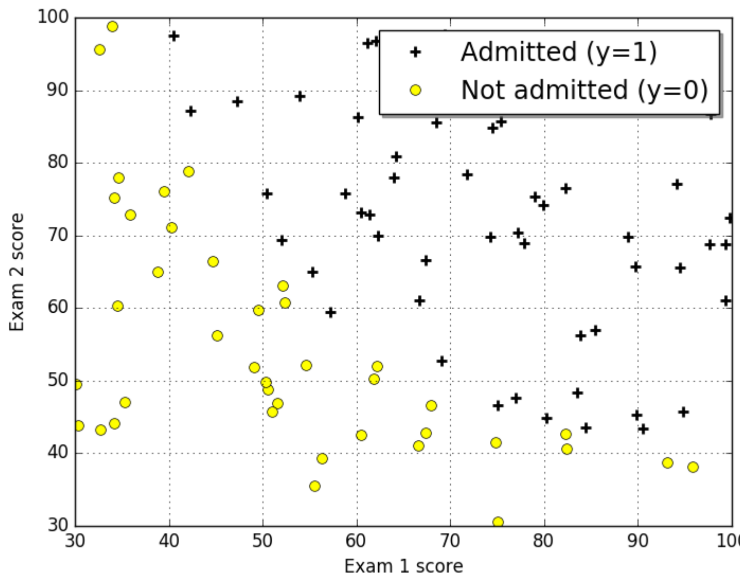

Figure 1 – Scatter plot of training data

To help you get more familiar with plotting, we have left **plotData.py** empty so you can try to implement it yourself. However, this is an optional exercise. We also provide our implementation below so you can copy it or refer to it. If you choose to copy our example, make sure you learn what each of its commands is doing by consulting the Python documentation.

```
pos = X[(y==1).flatten(),:]
neg = X[(y==0).flatten(),:]

plt.plot(pos[:,0], pos[:,1], '+', markersize=7, markeredgecolor='black', markeredgewidth=2)
plt.plot(neg[:,0], neg[:,1], 'o', markersize=7, markeredgecolor='black', markerfacecolor='yellow')

plt.legend(['Admitted (y=1)', 'Not admitted (y=0)'], loc='upper right', shadow=True, fontsize='x-large',
↪  numpoints=1)
plt.grid()
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
```

3.2 Implementation

3.2.1 Warmup exercise: sigmoid function

You can start by watching these videos: Hypothesis representation, Decision boundary.

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

$$h_\theta(x) = g\left(x^T\theta\right),$$

where function g is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1+e^{-z}}.$$

Your first step is to implement this function in **sigmoid.py** so it can be called by the rest of your program. When you are finished, try testing a few values by calling `sigmoid(x)` at the python command line. For large positive values of x, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating sigmoid(0) should give you exactly 0.5. Your code should also work with vectors and matrices. For a vector or a matrix, your function should perform the sigmoid function on every element.

3.2.2 Cost function and gradient

You can start by watching these videos: Cost function, Simplified cost function and gradient descent.

Now you will implement the cost function and gradient for logistic regression. Complete the code in **costFunction.py** and **gradientFunction.py** to return the cost and gradient. Recall that the cost function in logistic regression is

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right],$$

and the gradient of the cost is a vector where the j^{th} element (for $j = 0, 1, \ldots, n-1$) is defined as follows:

$$\frac{\partial J(\theta)}{\partial\theta_j} = \frac{1}{m}\sum_{i=0}^{m-1}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$.

Once you are done, **ex2.py** will call your **costFunction** using the initial parameters of θ. You should see that the cost is about 0.693.

3.2.3 Learning parameters using fmin_tnc

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly.

This time, instead of taking gradient descent steps, you will use a python built-in function called **fmin_tnc**.

Python's **fmin_tnc** is an optimization solver that finds the minimum of an unconstrained[1] function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ.

Concretely, you are going to use **fmin_tnc** to find the best parameters θ for the logistic regression cost function, given a fixed dataset (of X and y values). You will pass to **fmin_tnc** the following inputs:

- A function that, when given the training set and a particular θ, computes the logistic regression cost for the dataset (X, y)
- The initial values of the parameters we are trying to optimize.

[1] Constraints in optimization often refer to constraints on the parameters, for example, constraints that bound the possible values θ can take (e.g., $\theta \leq 1$). Logistic regression does not have such constraints since θ is allowed to take any real value.

- A function that, when given the training set and a particular θ, computes the logistic regression gradient with respect to θ for the dataset (X, y)

In **ex2.py**, we already have code written to call **fmin_tnc** with the correct arguments.

```
theta = opt.fmin_tnc(costFunction, initial_theta, gradientFunction, args=(X, y))
```

If you have completed the costFunction correctly, **fmin_tnc** will converge on the right optimization parameters and return the final values of the cost and θ. Notice that by using **fmin_tnc**, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. This is all done by **fmin_tnc**: you only needed to provide a function calculating the cost and the gradient.

Once **fmin_tnc** completes, **ex2.py** will call your **costFunction** function using the optimal parameters of θ. You should see that the cost is about 0.203.

This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2. We also encourage you to look at the code in **plotDecisionBoundary.py** to see how to plot such a boundary using the θ values.

3.2.4 Evaluating logistic regression

You can start by watching this video: The problem of overfitting.

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.774.

Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the code in **predict.py**. The predict function will produce 1 or 0 predictions given a dataset and a learned parameter vector θ.

After you have completed the code **predict.py**, the **ex2.py** script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

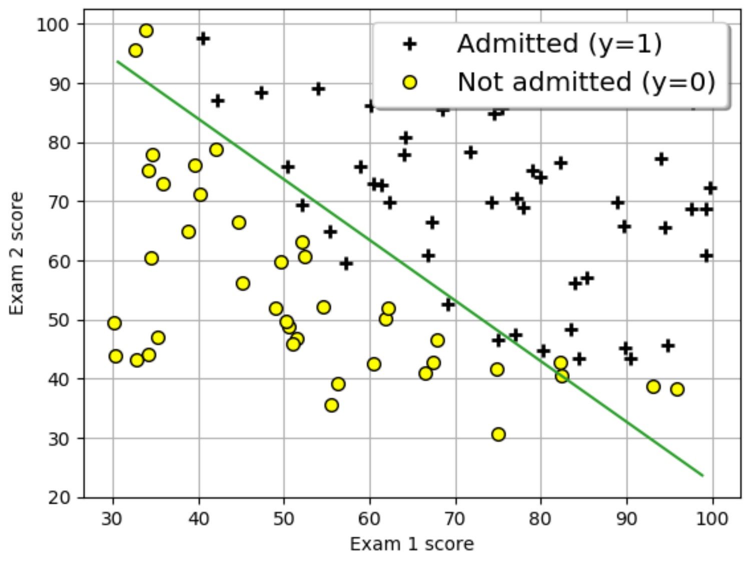

Figure 2 – Training data with decision boundary.

4 Regularized logistic regression

You can start by watching these videos: Regularized Logistic Regression, Regularized Logistic Regression.

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly.

Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

You will use another script, **ex2_reg.py** to complete this portion of the exercise.

4.1 Visualizing the data

Similar to the previous parts of this exercise, **plotData** is used to generate a figure like Figure 3, where the axes are the two test scores, and the positive ($y = 1$, accepted) and negative ($y = 0$, rejected) examples are shown with different markers.

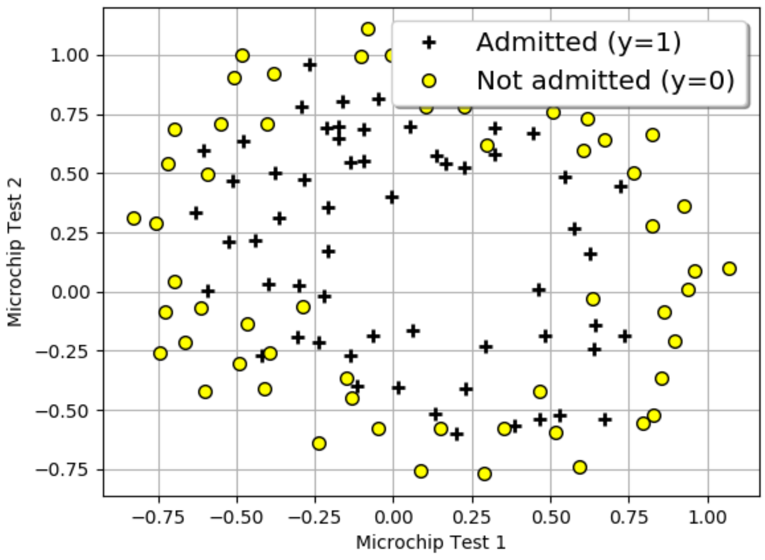

Figure 3 – Plot of training data.

Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight-forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

4.2 Feature mapping

One way to fit the data better is to create more features from each data point. In the provided function **mapFeature** in **plotDecisionBoundary.py**, we will map the features into all polynomial terms of x1 and x2 up to the sixth power.

$$mapFeature(x) = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{pmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot.

While the feature mapping allows us to build a classifier with more complex decision boundaries, it is also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

4.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Complete the code in **costFunctionReg.py** and **gradientFunctionReg.py** to return the cost and gradient.

Recall that the regularized cost function in logistic regression is

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n-1}\theta_j^2,$$

Note that you should not regularize the parameter θ_0; thus, the final summation above is for $j = 1$ to $n-1$, not $j = 0$ to $n-1$. The gradient of the cost function is a vector where the j-th element is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=0}^{m-1}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_0^{(i)} \qquad \text{pour } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\left(\sum_{i=0}^{m-1}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} + \lambda\theta_j\right) \qquad \text{pour } j \geq 1$$

Once you are done, **ex2_reg.py** will call your **costFunctionReg** function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

4.3.1 Learning parameters using fmin_tnc

Similar to the previous parts, you will use **fmin_tnc** to learn the optimal parameters θ. If you have completed the cost and gradient for regularized logistic regression (**costFunctionReg.py** and **gradientFunctionReg**) correctly, you should be able to step through the next part of **ex2_reg.py** to learn the parameters θ using **fmin_tnc**.

4.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function **plotDecisionBoundary.py** which plots the (non-linear) decision boundary that separates the positive and negative examples. In **plotDecisionBoundary.py**, we plot the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then draw a contour plot of where the predictions change from $y = 0$ to $y = 1$.

After learning the parameters θ, the next step in **ex_reg.py** will plot a decision boundary similar to Figure 5.

4.5 Impact of λ

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents overfitting.

Notice the changes in the decision boundary as you vary λ. With a small λ, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 4). This is not a good decision boundary: for example, it predicts that a point at $x = (-0.25; 1.5)$ is accepted ($y = 1$), which seems to be an incorrect decision given the training set.

With a larger λ, you should see a plot that shows an simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data (Figure 6).

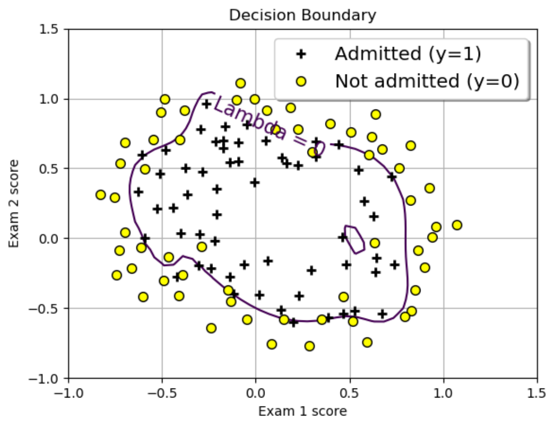

Figure 4 – No regularization (Overfitting) ($\lambda = 0$)

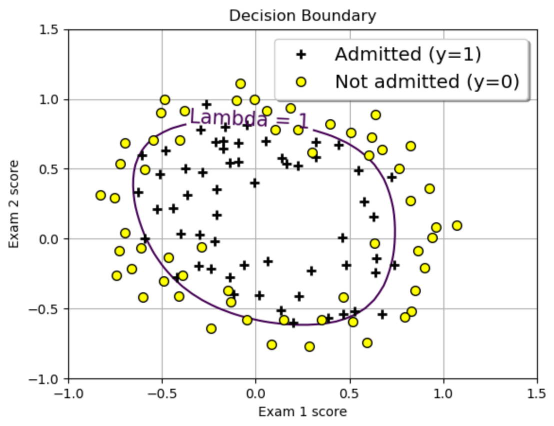

Figure 5 – Training data with decision boundary ($\lambda = 1$)

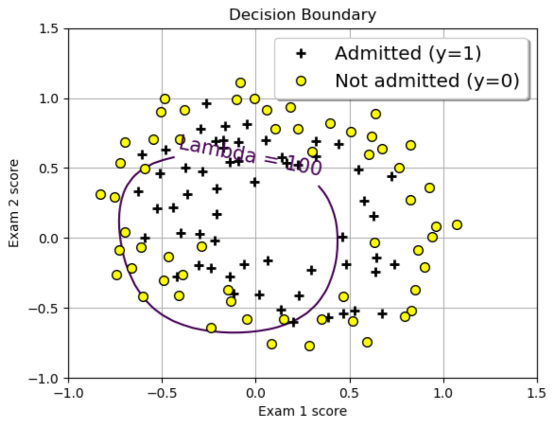

Figure 6 – Too much regularization (Underfitting) ($\lambda = 100$)

5 Multi-class classification

In this last part of the assignment, you will implement one-vs-all logistic regression to recognize handwritten digits.

You can start by watching this video: Multi-class classification One Vs All.

Throughout the exercise, you will be using the scripts **ex2_multiclass.py**. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify these scripts. You are only required to modify functions in other files, by following the instructions in this assignment.

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 in the first parts can be used for this classification task. You will extend your previous implemention of logistic regression and apply it to one-vs-all classification.

5.1 Dataset

You are given a data set in ex2data3.mat that contains 5000 training examples of handwritten digits. The .mat format means that the data have been saved in a native 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 **scipy.io.loadmat** command:

```
# Load saved matrices from file
datafile = 'ex2data3.mat'
mat = scipy.io.loadmat( datafile )
X, y = mat['X'], mat['y']
# X is an array (5000 x 400)
# y is an array (5000 x 1)
```

The matrices X and y will now be in your Python environment. There are 5000 training examples in ex2data3.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 become a single row in our data matrix X. This gives us a $m = 5000$ by $n = 400$ matrix X where every row is a training example for a handwritten digit image.

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

where $x^{(i)}_{i=1,\dots,m}$ is a n-dimensional vector. The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. **We have mapped the digit zero 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.**

5.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of **ex2_multiclass.py**, the code randomly selects 100 rows from X 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. We have provided the displayData function, 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 7.

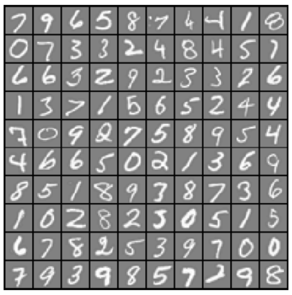

Figure 7 – Examples from the dataset.

5.3 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.

If in the previous parts, you have already computed a vectorized version (without any loops) of your regularized cost function and your regularized gradient function, you only need to copy their codes into **lrCostFunction.py** and **lrCostGradient.py**. Otherwise, you will find details on how to vectorize codes in the sequel.

5.3.1 Vectorizing the cost function

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

$$J(\theta) = \frac{1}{m}\sum_{i=0}^{m-1}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right].$$

To compute each element in the summation, we have to compute $h_\theta\left(x^{(i)}\right)$ for every example i, where $h_\theta\left(x^{(i)}\right) = g\left(\theta^T x^{(i)}\right)$ 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

$$X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \vdots \\ \left(x^{(m)}\right)^T \end{bmatrix} \text{ and } \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

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

$$X\theta = \begin{bmatrix} \left(x^{(1)}\right)^T\theta \\ \left(x^{(2)}\right)^T\theta \\ \vdots \\ \left(x^{(m)}\right)^T\theta \end{bmatrix} = \begin{bmatrix} \theta^T\left(x^{(1)}\right) \\ \theta^T\left(x^{(2)}\right) \\ \vdots \\ \theta^T\left(x^{(m)}\right) \end{bmatrix}$$

In the last equality, we used the fact that $a^Tb = b^Ta$ if a and b are vectors. This allows us to compute the products $\theta^T x^{(i)}$ for all our examples i in one line of code. Your job is to write the unregularized cost function in the file **lrCostFunction.py**. Your implementation should use the strategy we presented above to calculate $\theta^T x^{(i)}$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of **lrCostFunction.py** should not contain any loops. Hint: You might want to use the element-wise multiplication operation (*) and the sum operation when writing this function). The matrix multiplication operation is either @ or numpy.dot.

5.3.2 Vectorizing the gradient

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

$$\frac{\partial J(\theta)}{\partial\theta_j} = \frac{1}{m}\sum_{i=0}^{m-1}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

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

$$\begin{aligned}
\begin{bmatrix} \frac{\partial J(\theta)}{\partial\theta_0} \\ \frac{\partial J(\theta)}{\partial\theta_1} \\ \frac{\partial J(\theta)}{\partial\theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial\theta_n} \end{bmatrix} &= \frac{1}{m} \begin{bmatrix} \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)} \\ \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_1^{(i)} \\ \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_2^{(i)} \\ \vdots \\ \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_n^{(i)} \end{bmatrix} \\
&= \frac{1}{m} \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x^{(i)} \\
&= \frac{1}{m} X^T \left(h_\theta(x) - y\right) \qquad (1)
\end{aligned}$$

where

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

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

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

The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, we encourage you 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 **lrCostGradient.py** 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 100x20 (100 examples, 20 features) and, a vector with dimensions 20x1, you can observe that $X\theta$ is a valid multiplication operation, while θX 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.

5.3.3 Vectorizing regularized logistic regression

After you have implemented vectorization for logistic regression, you will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

$$J(\theta) = \frac{1}{m} \sum_{i=0}^{m-1} \left[-y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n-1} \theta_j^2.$$

Note that you should not be regularizing θ_0 which is used for the bias term. Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

$$\frac{\partial J(\theta)}{\partial\theta_0} = \frac{1}{m} \sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} \qquad \text{pour } j = 0$$

$$\frac{\partial J(\theta)}{\partial\theta_j} = \frac{1}{m} \left(\sum_{i=0}^{m-1} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \lambda\theta_j \right) \qquad \text{pour } j \geq 1$$

Now modify your code in **lrCostFunction** and in **lrCostGradient** to account for regularization. Once again, you should not put any loops into your code.

Python Tip: When implementing the vectorization for regularized logistic regression, you might often want to only sum and update certain elements of θ. In Python, you can index into the matrices to access and update only certain elements. For example, A[:, 3:5] = B[:, 1:3] will replaces the columns 4 to 5 of A with the columns 2 to 3 from B.

5.4 Learning a One-vs-all classifier

In this part of the exercise, you will discover the way to make an one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 7). In the handwritten digits dataset, $K = 10$, but your code should work for any value of K.

You should now read carefully the code in **learnOneVsAll.py** to train one classifier for each class. In particular, note the code return all the classifier parameters in a matrix $\Theta \in \mathbb{R}^{K\text{x}(n+1)}$, where each row of Θ corresponds to the learned logistic regression parameters for one class. This is achieved 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 we have mapped the digit "0" to the label 10 (to avoid confusions with indexing).

When training the classifier for class $k \in \{1, \ldots, K\}$, 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 note that logical operators is helpful for this task.

Python Tip: Logical arrays in Python are arrays which contain binary (true or false) elements. In Python, evaluating the expression $a == b$ for a vector a (of size mx1) 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 Python:

```
a = np.arange(0,9,1) # Create a and b
b = 3
p = (a == b) # You should try different values of b here
print(p)
```

After you have carefully examined the code for **learnOneVsAll.py**, the script **ex2_multiclass.py** will continue to use your learnOneVsAll function to train a multiclass classifier.

5.5 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.py** to use the one-vs-all classifier to make predictions.

Once you are done, **ex2_multiclass.py** will call your predictOneVsAll function using the learned value of Θ. You should see that the training set accuracy is about 96.5% (i.e., it classifies 96.5% of the examples in the training set correctly).

5.6 Some questions, you have to answer...

Le codage n'est qu'un prétexte pour comprendre les différents concepts du machine learning. **Ici aucune équation...Exprimez avec vos mots les concepts pour les comprendre.**

1. Identifiez les différences d'approche entre le TP1 et le TP2? et les différences de résolution du problème?

2. A quoi sert le principe de régularisation ?

3. Décrivez le problème du sur-apprentissage et comment on y répond dans ce TP. Quelle est l'influence de la valeur de λ?

4. Dans quelles circonstances et pour quelles raisons utilise-t-on l'approche multiclasse ? Quelles sont les différentes possibilités pour cette approche?