Programming Exercise 3: Logistic Regression

Files included in this exercise

File	Description
`data/ex2data1.txt`	Training set for the first half of the exercise
`data/ex2data2.txt`	Training set for the second half of the exercise
`public_tests.py`	Functions to test cost and gradient calculation
`utils.py`	Utility functions
`test_utils.py`	Utility functions for tests
[⋆] `logistic_reg.py`	Functions to compute the cost and the gradient of logistic regression and to run gradient descent

[⋆] indicates files you will need to complete

Part A: logistic regression

Problem Statement

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.

Training data is shown in Figure 1.1 which has been generated with the help of function `utils.plot_data`:

```
def plot_data(X, y, pos_label="y=1", neg_label="y=0"):
    positive = y == 1
    negative = y == 0

    plt.plot(X[positive, 0], X[positive, 1], 'k+', label=pos_label)
    plt.plot(X[negative, 0], X[negative, 1], 'yo', label=neg_label)
```

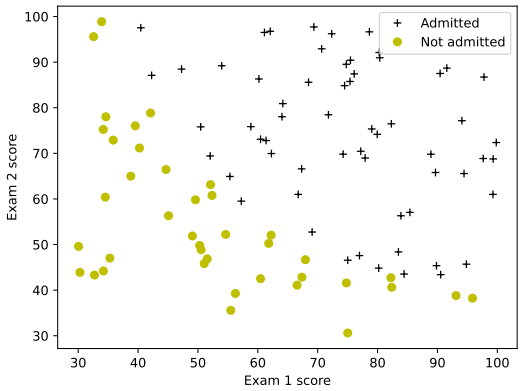

Figure 1.1: Dataset

Your task is to build a classification model that estimates an applicant's probability of admission based on the scores from those two exams.

Sigmoid function

The sigmoid function is defined as:

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

Implement the sigmoid function first, so it can be used in the rest of this exercise:

```
def sigmoid(z):
```

```
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    return g
```

You can check if your implementation is correct by computing the sigmoid of $[-1, 0, 1, 2]$ which should return $[0.26894142, 0.5, 0.73105858, 0.88079708]$. In addition you should also run and pass the tests from `public_tests.sigmoid_test`.

Compute cost

The equation for the cost function for logistic regression is:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) \right]$$

where:

$$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log \left(f_{\mathbf{w},b} \left(\mathbf{x}^{(i)} \right) \right) - \left(1 - y^{(i)} \right) \log \left(1 - f_{\mathbf{w},b} \left(\mathbf{x}^{(i)} \right) \right)$$

and $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{i})} + b)$ where function g is the sigmoid function.

Complete the `logistic_reg.compute_cost` function to implement the computation of the cost[1]:

```
def compute_cost(X, y, w, b, lambda_=None):
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
```

[1]The regularization parameter λ is denoted as the variable name `lambda_`, notice the underscore at the end of the name. This is because `lambda` is a reserved python keyword, and should never be used as a variable name.

```
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : scalar Values of bias parameter of the model
      lambda_: unused placeholder
    Returns:
      total_cost: (scalar)         cost
    """

    return total_cost
```

You can check if your implementation is correct by computing the cost with some initial values for parameters w and b. For example, with $\vec{w}$ and b set to 0 the expected output of the cost function is 0.693, while $w = [0.2, 0.2]$ and $b = -24$ should result in 0.218. In addition you should also run and pass the tests from `public_tests.compute_cost_test`.

Compute gradient

The gradient for logistic regression with multiple variables is defined as:

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}$$
$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

Complete the `logistic_reg.compute_gradient` function to implement the computation of the gradient:

```
def compute_gradient(X, y, w, b, lambda_=None):
    """
    Computes the gradient for logistic regression

    Args:
      X : (ndarray Shape (m,n)) variable such as house size
      y : (array_like Shape (m,1)) actual value
      w : (array_like Shape (n,1)) values of parameters of the model
      b : (scalar)                 value of parameter of the model
      lambda_: unused placeholder.
    Returns
      dj_db: (scalar)                The gradient of the cost w.r.t. the
          parameter b.
```

```
13        dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the
              parameters w.
14      """
15
16      return dj_db, dj_dw
```

You can check if your implementation is correct by computing the gradient with some initial values for parameters w and b. For example, with $\vec{w}$ and b set to 0 the expected output of the gradient function is -0.1 for `dj_db` and $[-12.00921658929115, -11.262842205513591]$ for `dj_dw`, while with $w = [0.2, -0.5]$ and $b = -24$ the expected output of the gradient function is -0.5999999999991071 for `dj_db` and $[-44.8313536178737957, -44.37384124953978]$ for `dj_dw`. In addition you should also run and pass the tests from `public_tests.compute_gradient_test`.

Gradient descent

The gradient descent algorithm is:

$$\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\quad w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \quad \text{for j = 0..n-1} \\
&\quad b \;\; = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&\}
\end{aligned}$$

where n is the number of features, and parameters w_j, b are updated simultaniously.

Complete the `logistic_reg.gradient_descent` function to implement the batch gradient descent algorithm:

```
1  def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function,
       alpha, num_iters, lambda_=None):
2      """
3      Performs batch gradient descent to learn theta. Updates theta by taking
4      num_iters gradient steps with learning rate alpha
5
6      Args:
7        X :    (array_like Shape (m, n)
8        y :    (array_like Shape (m,))
```

```
9         w_in : (array_like Shape (n,))  Initial values of parameters of the
              model
10        b_in : (scalar)                 Initial value of parameter of the model
11        cost_function:                  function to compute cost
12        alpha : (float)                 Learning rate
13        num_iters : (int)               number of iterations to run gradient
              descent
14        lambda_ (scalar, float)         unused placeholder
15
16    Returns:
17        w : (array_like Shape (n,)) Updated values of parameters of the model
18            after running gradient descent
19        b : (scalar)                Updated value of parameter of the model
20            after running gradient descent
21        J_history : (ndarray): Shape (num_iters,) J at each iteration,
22            primarily for graphing later
23    """
24
25    return w, b, J_history
```

You can now run your implementation of gradient descent initializing b to -8 and $\vec{w}$ to 0. and running it 10000 iterations with a learning rate $\alpha = 0.001$ which should get a cost around 0.30. It may take a couple of minutes to run. You can reduce the iterations to test your implementation and iterate faster. If you have time, try running 100,000 iterations for better results.

We will now use the final parameters from gradient descent to plot the linear fit. If you implemented the previous parts correctly, you should see the plot shown in Figure 1.2 which has been generated with the function `utils.plot_decision_boundary`.

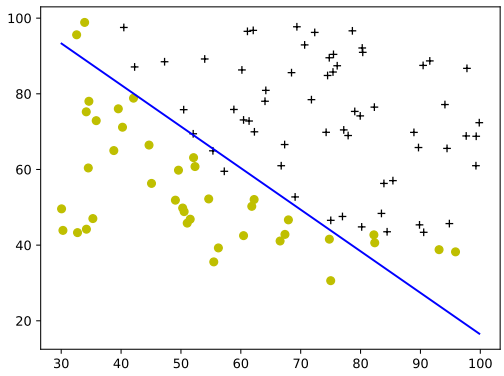

Figure 1.2: Linear fit

Evaluating logistic regression

We can evaluate the quality of the parameters we have found by seeing how well the learned model predicts on our training set. Implement the `logistic_reg.predict` function below to do this.

```
def predict(X, w, b):
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w and b

    Args:
    X : (ndarray Shape (m, n))
    w : (array_like Shape (n,))      Parameters of the model
    b : (scalar, float)              Parameter of the model

    Returns:
    p: (ndarray (m,1))
        The predictions for X using a threshold at 0.5
    """

    return p
```

You can check if your implementation is correct by running and passing the tests from `public_tests.predict_test`.

Finally, you can compute the percentage of training examples that the final parameters from gradient descent correctly classify. The accuracy on the training set should be around 92.00%.

Part B: regularized logistic regression

Problem Statement

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.

Training data is shown in Figure 1.3 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. This plot has been generated with the help of function `utils.plot_data`:

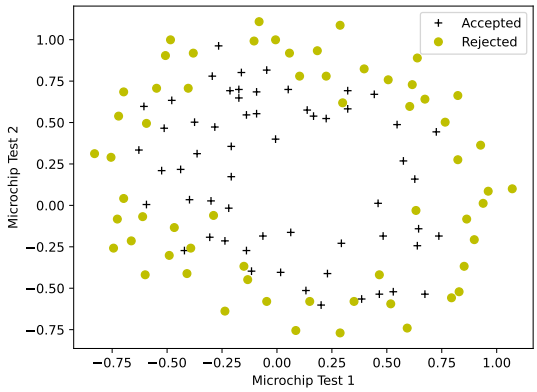

Figure 1.3: Plot of training data

Feature mapping

Figure 1.3 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.

One way to fit the data better is to create more features from each data point. In the provided function `utils.map_feature`, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

$$\text{map\_feature}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 27-dimensional vector that we will use as training data in the rest of this exercise.

Compute cost for regularized logistic regression

For regularized logistic regression, the cost function is of the form:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[-y^{(i)} \log \left(f_{\mathbf{w},b} \left(\mathbf{x}^{(i)} \right) \right) - \left(1 - y^{(i)} \right) \log \left(1 - f_{\mathbf{w},b} \left(\mathbf{x}^{(i)} \right) \right) \right] + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

Complete the `logistic_reg.compute_cost_reg` function to implement the computation of the cost for regularized logistic regression:

```
def compute_cost_reg(X, y, w, b, lambda_=1):
    """
    Computes the cost over all examples
    Args:
      X : (array_like Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : (array_like Shape (n,)) Values of bias parameter of the model
      lambda_ : (scalar, float)    Controls amount of regularization
```

```
    Returns:
      total_cost: (scalar)         cost
    """

    return total_cost
```

You can check if your implementation is correct by running and passing the tests from `public_tests.compute_cost_reg_test`.

Gradient for regularized logistic regression

The gradient of the regularized cost function is defined as follows:

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \left(\frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}w_j \quad \text{for } j = 0...(n-1)$$

Complete the `logistic_reg.compute_gradient_reg` function to calculate the gradient for the regularized cost function.

```
def compute_gradient_reg(X, y, w, b, lambda_=1):
    """
    Computes the gradient for linear regression

    Args:
      X : (ndarray Shape (m,n))   variable such as house size
      y : (ndarray Shape (m,))    actual value
      w : (ndarray Shape (n,))    values of parameters of the model
      b : (scalar)                value of parameter of the model
      lambda_ : (scalar,float)    regularization constant
    Returns
      dj_db: (scalar)             The gradient of the cost w.r.t. the
          parameter b.
      dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the
          parameters w.

    """

    return dj_db, dj_dw
```

You can check if your implementation is correct by running and passing the tests from `public_tests.compute_gradient_reg_test`.

Learning parameters using gradient descent

You can now use the gradient descent algorithm implemented in the first half of the exercise with the regularized versions of cost and gradient computation to learn the optimal parameters w, b. Initializing b to 1 and $\vec{w}$ to 0, with $\lambda = 0.01$, $alpha = 0.01$ and 10,000 iterations you should obtain a cost below 0.5. It may take quite a while to run. You can reduce the iterations to test your implementation and iterate faster. If you have time, try running 100,000 iterations for better results.

We will now use the final parameters from gradient descent to plot the non-linear decision boundary. If you implemented the previous parts correctly, you should see the plot shown in Figure 1.4 which has been generated with the function `utils.plot_decision_boundary`.

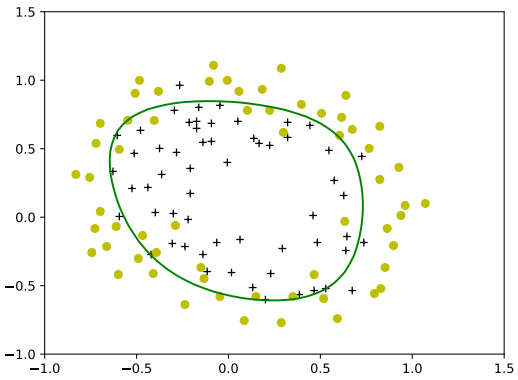

Figure 1.4: Non-linear decision boundary

Finally, you will use the `logistic_reg.predict` function that you implemented before to calculate the accuracy of the regulaized logistic regression model on the training set, which should be around 80.00%