Create – Applications From Ideas
Written Response Submission Template

Please see Assessment Overview and Performance Task Directions for Student for the task directions and recommended word counts.

Program Purpose and Development

2a)

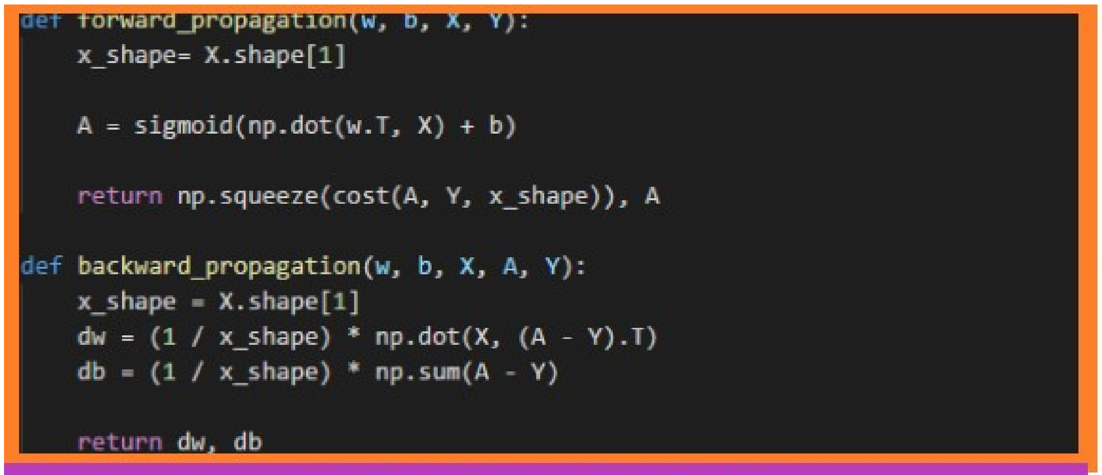
Explanation is included in video.

2b)

During the development of the logistic regression model, I encountered problems with multiplying the array of pixels. When multiplying, if the array sizes do not match, an error occurs and stops code execution. Initially, I did not realize that all images must be the same size to match the weights and biases. To rectify, I had to determine the standard size of the images. After research, I decided 64 pixels would give the algorithm enough accuracy to differentiate between colors and shapes.

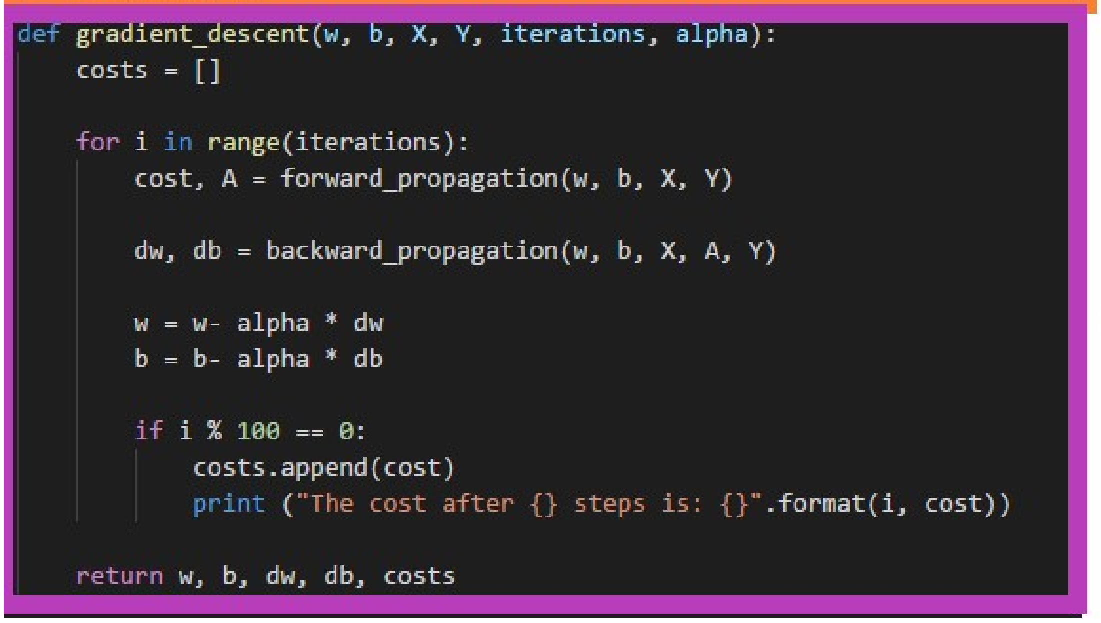
The second impediment was determining hyperparameters including the number of iterations and the learning rate. This process was mostly experimental as the outcome cannot be determined beforehand. I experimented with approximately 15 different sets of hyperparameters before concluding with 2,000 iterations and a learning rate of 0.005. The learning rate was relatively low because the model is relatively simplistic. After 2,000 iterations, the model accuracy hit a plateau. Not having an experienced peer to consult made the program more difficult to debug.

2c)



```
def forward_propagation(w, b, X, Y):
    x_shape= X.shape[1]

    A = sigmoid(np.dot(w.T, X) + b)

    return np.squeeze(cost(A, Y, x_shape)), A

def backward_propagation(w, b, X, A, Y):
    x_shape = X.shape[1]
    dw = (1 / x_shape) * np.dot(X, (A - Y).T)
    db = (1 / x_shape) * np.sum(A - Y)

    return dw, db
```

#1



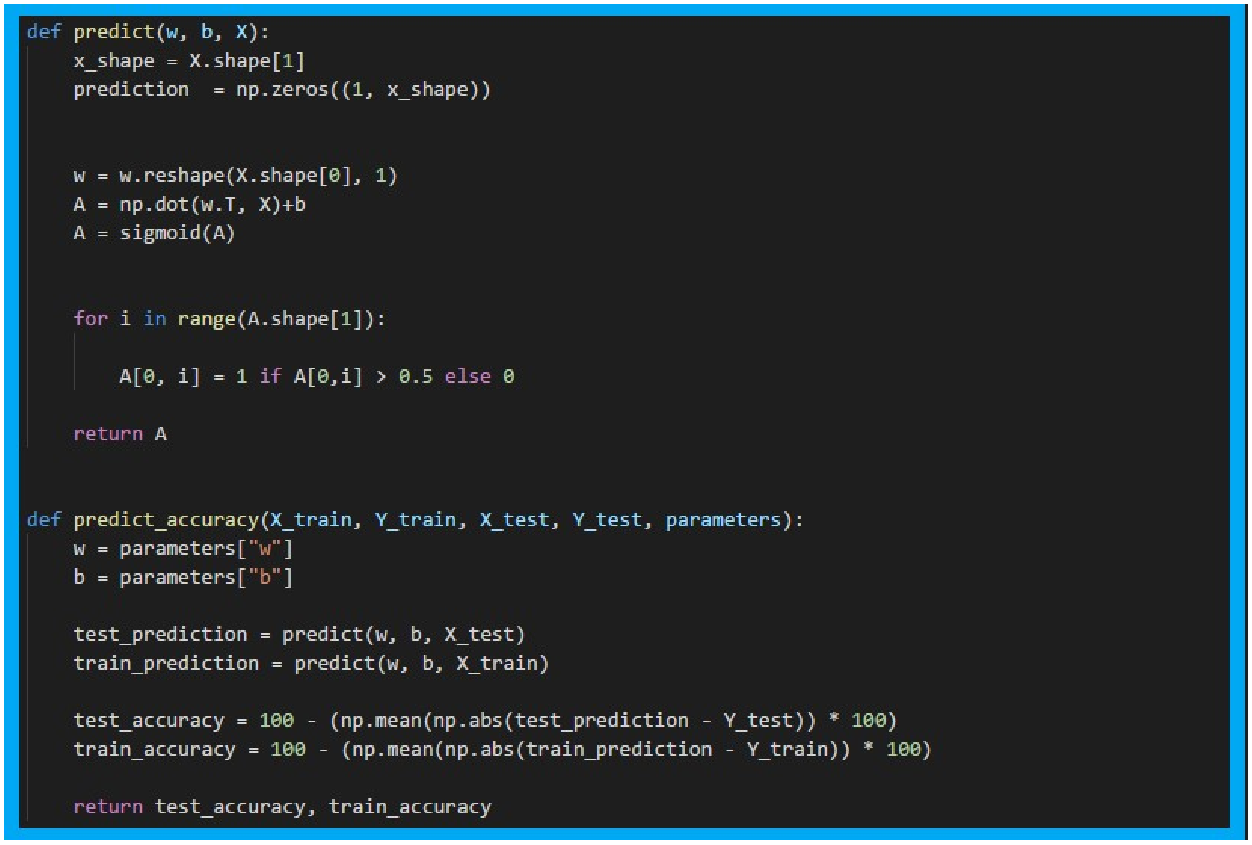
```
def gradient_descent(w, b, X, Y, iterations, alpha):
    costs = []

    for i in range(iterations):
        cost, A = forward_propagation(w, b, X, Y)

        dw, db = backward_propagation(w, b, X, A, Y)

        w = w- alpha * dw
        b = b- alpha * db

        if i % 100 == 0:
            costs.append(cost)
            print ("The cost after {} steps is: {}".format(i, cost))

    return w, b, dw, db, costs
```

#2

I built a logistic regression model on two calculations of the data, defined in the functions forward_propagation and backward_propagation (shown above #1). These two algorithms are essential for the program to work because they control the learning parameters that are used to update the weights and biases. The forward propagation step analyzes the shape of the input data, which is essential to eliminate errors in the array multiplication. The forward propagation step then computes the sigmoid function of a linear equation which multiplies the current transposed weights by the input data using the np.dot function, then adds the bias. The algorithm returns the cost function and the result of the sigmoid function. The next step is back propagation which takes the result of the sigmoid function and calculates the partial derivatives of the weights and biases. The algorithm is called back propagation as it needs to move backwards to calculate the partial derivatives of the weights and biases. This is essential because the weights and biases are updated from the partial derivatives calculated in the previous step.

The last integrated algorithm implemented into the code (shown above #2) is the gradient descent function which puts the previous two algorithms together to update the weights and biases. This algorithm updates the weights and biases by first running forward propagation to compute the cost and the result of the sigmoid function(A). Then, using those parameters, it runs backward propagation to calculate the derivatives (dw and db). This runs based on how many specified iterations there are in the iterations section of the input. The gradient descent function returns the final weights and biases along with the final derivatives of w and b. The implemented algorithms can run independently to compute forward and backward propagation. This algorithm helps to streamline the process of using forward and backward propagation as, without this, retraining of the model could not be done by directly calling a function.

2d)



```
def predict(w, b, X):
    x_shape = X.shape[1]
    prediction  = np.zeros((1, x_shape))


    w = w.reshape(X.shape[0], 1)
    A = np.dot(w.T, X)+b
    A = sigmoid(A)


    for i in range(A.shape[1]):

        A[0, i] = 1 if A[0,i] > 0.5 else 0

    return A


def predict_accuracy(X_train, Y_train, X_test, Y_test, parameters):
    w = parameters["w"]
    b = parameters["b"]

    test_prediction = predict(w, b, X_test)
    train_prediction = predict(w, b, X_train)

    test_accuracy = 100 - (np.mean(np.abs(test_prediction - Y_test)) * 100)
    train_accuracy = 100 - (np.mean(np.abs(train_prediction - Y_train)) * 100)

    return test_accuracy, train_accuracy
```

Though most of the algorithms shown above could be labeled as abstraction since they simplify the process of performing calculations multiple times, a separate example is the prediction function (shown above). This abstraction was developed in order to manage the method of predicting a piece of data without doing specific nitty-gritty tasks, including determining the shape of the prediction candidate data. Without this abstraction, to test the accuracy of data using the predict accuracy function (shown above), shapes of data would have to be explicitly defined and the operations included in prediction would be called multiple times throughout the script including testing training data, test data and any other data. This abstraction saves multiple redundancies that would generally happen without it. Thus, improving readability and ease of use.