Gradient Descent Exercise

due September 30

CSMC 352: Machine Learning
Prof. Rose Sloan
Nathan Cho
nc5123@bard.edu

Part I

Implementation

```
def compute_grad(data, weights):
    derivs = []

    for x, y in data:
        hx = weights[0] + weights[1] * x
        derivs.append(np.array([hx - y, x * (hx - y)]))

    return np.sum(derivs)  # return sum of derivatives
```

According to the batch gradient descent algorithm, the derivatives are added up for all data points in each iteration.

Result

```
Iteration 1
Weights are currently <0.007630, 0.909380>
Current cost: 12276.569740
Iteration 2
Weights are currently <0.002310, 0.267722>
Current cost: 6215.537392
...
Iteration 24
Weights are currently <0.005340, 0.533043>
Current cost: 205.929103
Iteration 25
Weights are currently <0.005379, 0.533253>
Current cost: 205.928311
```

There were 25 iterations and the weights were $w_0 = 0.005379$, $w_1 = 0.533253$.
Cost of the final iteration was 205.928311.

Part 2

Implementation

```
def compute_grad(data, weights):
    deriv = None

    for x, y in data:
        hx = weights[0] + weights[1] * x
        deriv = np.array([hx - y, x * (hx - y)])
        weights -= lr * deriv

    return deriv  # return last derivative
```

According to the stochastic gradient descent algorithm, the derivatives are immediately updating the weights for all data points in each iteration.

Result

```
Iteration 1
Weights are currently <0.004051, 0.471734>
Current cost: 527.864487
Iteration 2
Weights are currently <0.004616, 0.533686>
Current cost: 205.953051
Iteration 3
Weights are currently <0.004722, 0.541822>
Current cost: 212.312427
```

There were 3 iterations and the weights were $w_0 = 0.004722$, $w_1 = 0.541822$.
Cost of the final iteration was 212.312427.

Batch v. Stochastic

Benefits of using batch gradient descent algorithm is that while being same in computational complexity as stochastic algorithm, batch algorithm will generally approximate true minimum better.

Benefits of using stochastic gradient descent algorithm is that because the weights are updated during the iteration, it will reach the approximation substantially faster. On the other hand, it is likely to never converge, so it's necessary to use a stopping criteria with this algorithm.

Additionally, As shown in iteration 3 above, it is possible that the cost increases. In order to prevent stopping at a higher weight, we can add the following highlighted code.

```
while cost_decrease > 0.001:
    deriv = compute_grad(all_data, w)
    w -= lr * deriv
    newcost = compute_cost(all_data, w)
    cost_decrease = cost - newcost

    if cost_decrease <= 0:
        break

    cost = newcost
    print_info(i, w, cost)
    i += 1
```

With this additional stopping criteria, the results can be improved.

```
Iteration 1
Weights are currently <0.004051, 0.471734>
Current cost: 527.864487
Iteration 2
Weights are currently <0.004616, 0.533686>
Current cost: 205.953051
```

There were 2 iterations and the weights were w_0 = 0.004616, w_1 = 0.533686.
Cost of the final iteration was 205.953051.