COMMON PITFALLS: HW 1, PART 3

1. About this document

As part of an ongoing research project to provide high quality autonomous feedback in online courses, we are making this list of common errors from Homework 1 available to all current students. This list was generated by automatically mining "exemplar submissions" from hundreds of thousands of submissions in the previous iteration of this course. If you have any questions, complaints or general feedback, please email codewebresearch@gmail.com. And stay tuned for the release of our interactive feedback tool in a future homework!

2. Common Errors

Error 1. (Normal equations instead of gradient descent)
Instead of doing gradient descent, many people tried to set θ via the normal equations. While this gives a "technically" correct value for θ, it is *not* gradient descent.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    theta = pinv (X' * X) * X' * y
    J_history (iter) = computeCost (X, y, theta);
```

Error 2. (The extraneous sum)
Another common error was to include an extraneous sum in the θ update as shown below, which made the updates to both components of θ identical. In the following example of this, one would also need to transpose the expression, e.g., $((X * \theta - y)' * X)'$.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    theta = theta - alpha / m * sum ((X * theta - y)' * X)
    J_history (iter) = computeCost (X, y, theta);
```

Error 3. (Asynchronous updates for θ)

This approach was almost correct and would have been correct if the θ updates had been put into temporary variables first, then updated in a third step simultaneously. In the following example implementation, $\theta(2)$ is updated according to the new value of $\theta(1)$ rather than the $\theta(1)$ from the previous iteration.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    theta (1) = theta (1) - alpha / m * sum ((X * theta - y) .* X (:, 1))
    theta (2) = theta (2) - alpha / m * sum ((X * theta - y) .* X (:, 2))
    J_history (iter) = computeCost (X, y, theta);
```

Error 4. (Asynchronous updates with respect to training examples)

This approach is essentially what is known as stochastic gradient descent (the homework asks for a batch implementation). The problem in the following example is that for each datapoint the submitted code updates based on the theta that was obtained based on previous datapoints in the sequence.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    for i = 1:m
      temp1 = theta (1) - alpha / m * (theta (1) + theta (2) * X (i, 2) - y (i));
      temp2 = theta (2) - alpha / m * (theta (1)
                   + theta (2) * X (i, 2) - y (i)) * X (i, 2);
      theta (1) = temp1;
      theta (2) = temp2;
    endfor;
    J_history (iter) = computeCost (X, y, theta);
```

Error 5. (Ignoring the $X(:,1)$ column)

A significant number of people ignored the X(:,1) column and effectively ended up updating both components of θ in the same way.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    theta = theta - alpha / m * sum ((X * theta - y) .* X (:, 2))
    J_history (iter) = computeCost (X, y, theta);
```

Error 6. (Forgetting to divide gradient by m)

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    theta = theta - alpha * X' * (X * theta - y)
    J_history (iter) = computeCost (X, y, theta);
```

Error 7. (Updating with computeCost instead of gradient)
A significant number of people also tried to subtract computeCost at each iteration instead of the gradient.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    temp0 = theta (1) - alpha * computeCost (X, y, theta);
    temp1 = theta (2) - alpha * computeCost (X, y, theta);
    theta (1) = temp0;
    theta (2) = temp1;
    J_history (iter) = computeCost (X, y, theta);
```

Error 8. (Running to convergence at every iteration)
Some people ran gradient descent to convergence at each of *num_iters* iterations.

```
function [theta, J_history] = gradientDescent (X, y, theta, alpha, num_iters)
  m = length (y);
  J_history = zeros (num_iters, 1);
  for iter = 1:num_iters
    temp1 = theta (1);
    temp2 = theta (2);
    while true
      prevTemp1 = temp1;
      prevTemp2 = temp2;
      temp1 = theta (1) - alpha * (1 / m) * ((X * theta) - y)' * X (:, 1);
      temp2 = theta (2) - alpha * (1 / m) * ((X * theta) - y)' * X (:, 2);
      theta (1) = temp1;
      theta (2) = temp2;
      if abs (prevTemp1 - temp1) < 0.0001 && abs (prevTemp2 - temp2) < 0.0001
        break;
      endif;
    endwhile;
    J_history (iter) = computeCost (X, y, theta);
```