Week 2 Notes

Environment Setup Instructions

I'm using Windows 10 and VSCode for most things. Octave 6.1.0 was installed using the `.exe` file.

Octave 6.1.0 Documentation

Math and equations are written using KaTeX notation.

Multivariate Linear Regression

Multiple Features

Let's add some more features to our price prediction model for homes in Portland, OR. Here's an example of the data table we will use:

size ($feet^2$)	# of bedrooms	# of floors	age of home ($years$)	price ($1000)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

Some notation:

$$
\begin{aligned}
n &= \text{number of features} \\
x^{(i)} &= \text{input (features) of } i^{th} \text{ training example} \\
x_j^{(i)} &= \text{value of feature } j \text{ in } i^{th} \text{ training example}
\end{aligned}
$$

For example:

$$
\begin{aligned}
n &= 4 \\
x^{(2)} &= \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix} \\
x_2^{(3)} &= 2
\end{aligned}
$$

The hypothesis for this problem is now: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$

For convenience of notation, we define $x_0 = 1$, *i.e.*, $x_0^{(i)} = 1$.

$$
x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}
$$

This allows us to write:

$$
h_\theta(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x
$$

Gradient Descent for Multiple Variables

hypothesis: $h_\theta(x) = \theta^T x$

parameters: $\vec{\theta}$

cost function: $J(\vec{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right)^2$

gradient descent:

$$
\begin{aligned}
&\text{repeat } \{ \\
&\qquad \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \cdots, \theta_n) = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)}) \right) x_j^{(i)} \\
&\qquad \}
\end{aligned}
$$

Gradient Descent in Practice I - Feature Scaling

One trick to make multivariate gradient descent fitting work better is to make sure that the features are on the same scale. If they are very different, the contours of the fitting curve will be very narrow, making it difficult to asses which direction to vary the parameters to get to a minimum.

In practice, you can just normalize each feature (divide each value by the maximum feature value). This scales everything to $-1 \leq x_i \leq 1$.

You also want to avoid very small values for the features.

Another technique is to apply *mean normalization*, in which x_i is replaced with $x_i - \mu_i$ (except for $x_0 = 1$), and then divided by the maximum value.

More generally:

$x \leftarrow \frac{x_1 - \mu_1}{s_1}$

where μ_1 is the average value of x_1 in the training set and s_1 is the range of values ($x_{1,max} - x_{1,min}$)

Gradient Descent in Practice II - Learning Rate

One way to "debug" the process of gradient descent fitting is to plot $J(\vec{\theta})$ *vs*. the number of iterations of the algorithm. If things are working, this should be a curve that decreases with the number of iterations. Eventually, the curve should flatten as the values converge.

You can test for convergence by defining some minimum threshold value for the change in $J(\vec{\theta})$ in one iteration.

If the value of $J(\vec{\theta})$ is *increasing* with the number of iterations, this is often an indication that α should be smaller.

When troubleshooting values of α, vary the value by orders of magnitude and see how the convergence changes.

Features and Polynomial Regression

Features can be combined or modified as needed to change the hypothesis function. For example, say we want to produce a fit for the price of a house *vs*. its size, but we know that we don't necessarily want a linear or quadratic function, but instead a polynomial with order 3. The general form of the hypothesis function for this problem would then be:

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$

where $x_1 = (size)$, $x_2 = (size)^2$, and $x_3 = (size)^3$.

When using this method, feature scaling becomes increasingly important, as the size of each feature grows exponentially.

One can imagine other reasonable choices for a hypothesis function that uses other values for the exponenents, including fractional ones.

Computing Parameters Analytically

Normal Equation

A Normal Equation will allow us to solve for θ analytically.

If 1D $(\theta \in \mathbb{R})$ and $J(\theta) = a\theta^2 + b\theta + c$.

To minimize θ:

$\frac{d}{d\theta} J(\theta) = \cdots \stackrel{set}{=} 0$

and solve for θ.

When θ isn't just a scalar quantity, but a vector $(\theta \in \mathbb{R}^{n+1})$ and $J(\theta_0, \theta_1, \cdots, \theta_m) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)}\right)^2$, a similar approach can be taken:

$\frac{\partial}{\partial \theta_j} J(\theta) = \cdots \stackrel{set}{=} 0 \quad \text{for every } j$

And then solve for $\theta_0, \theta_1, \cdots, \theta_m$

The derivation of the solutions is somewhat involved.

Let's use the sample training dataset discussed in the Multiple Features section above. We'll also just cut off the extended table and only use the first four rows of the table so that $m = 4$.

First, construct the feature matrix (where we have specifed $x_0 = 1$ for all rows):

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$$

Next build the solution vector:

$$\vec{} \quad \vec{}$$

$$y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

The value of θ that minimizes y can then be computed as:

$$\theta = (X^T X^{-1}) X^T y$$

So for m *examples* $(x^{(1)}, y^{(1)}, \cdots, (x^{(m)}, y^{(m)})$ and n *features*:

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1}, \quad X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix}$$

where X is called the "design matrix," (or "model" or "regressor") and it has dimensions $m \times (n+1)$.

To solve for θ using Octave:

```
pinv(X'*X)*X'*y
```

Note: feature scaling is less important when using the normal equation to solve for optimum values of θ.

Pros and Cons of gradient descent *vs*. normal equation (for m training examples and n features):

Gradient Descent	**Normal Equation**
- need to choose α	- don't need to choose α
- needs many iterations	- no iterations
- works well even for large n	- need to compute $(X^T X)^{-1},\ O(n^3)$

Normal Equation Non-invertibility

What if $X^T X$ is non-invertible? The function `pinv()` (pseudo-inverse) in Octave works regardless.

One thing that can cause $X^T X$ to be non-invertible is having features that are linearly dependent on each other (redundant). Another problem can arise when the number of training examples is less than the number of features.

Submitting Programming Assignments

Working on and Submitting Assignments

This section is irrelevant to the online Coursera class.

Octave/Matlab Tutorial

Basic Operations

basic-operations.m

Moving Data Around

moving-data.m (loads data from featuresX.dat and pricesy.dat).

Computing on Data

computing.m

Plotting Data

Examples from the course: plotting.m

Octave Plotting Docs

Control Statements

Examples from the course: control.m

To use custom functions, save a file as `functionName.m` and then navigate to its directory. For example, the file `squareThisNumber.m` can be called with a single arugment (must be a number) and it will return the square of that number.

Functions can also return multiple values.

Here is an example of defining a cost function:

```
function J = costFunctionJ(X, y, theta)

% X is the design matrix containing training examples
% y is the class labels

m = size(X,1);                      % number of training examples
predictions = X*theta;              % predictions of hypothesis on all m
sqrErrors = (predictions - y).^2;  % squared errors

J = 1/(2*m) * sum(sqrErrors);
```

Vectorization

The typical hypothesis for linear regression:

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

Recall that this can also be represented as $h_\theta(x) = \theta^T x$.

For an *unvectorized* implementation, use the first representation:

```
prediction = 0.0;
for j = 1:n+1,  % from 1 to n+1 because octave indexes from 1, not 0
  prediction = prediction + theta(j)*x(j)
end;
```

In contrast, a *vectorized* implementation uses the matrix definition:

```
prediction = theta' * x;
```

The second implementation will run much more efficiently, and is simpler to represent.

Another example using gradient descent. Recall the update rules for θ (for $n = 2$ features):

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right) x_0^{(i)}$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right) x_1^{(i)}$$
$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right) x_2^{(i)}$$

To vectorize, we think of θ as a vector, and our update rules become $\theta := \theta - \alpha\delta$, where δ is also a vector (and is just the sum term from the equations above). So $\delta_0 = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y^{(i)} \right) x_0^{(i)}$,

etc.