Personal notes on Machine Learning Course - Coursera

Pedro B. Castro

November 21, 2017

Contents

Personal

This are my personal notes taken from Andrew Ng. course at Coursera. For explicit explanations, please refer to the course. This are for personal use only

Chapter 1

Week 1 and 2 - Regression

1.1 Concepts of Machine Learning

1. Definition of Machine Learning: Hability of computers to learn without being explicit programmed.
2. Definition of Supervised Learning: Learning from ”known” examples, ”right answers” and them predicting values for unknown inputs. Example: Given a dataset of values of houses with their respective sizes in a certain region, try to learn and predict value of a new house given their respective size.
3. Definition of Unsupervised Learning: Finding structure in data without knowing the effects of the variables. Such as clustering of stars, clustering of genomes etc..

1.2 Modeling and the Cost Function for one feature

Types of Supervised Learning

1. Regression: Continuable variables (house prices, etc..)

2. Classification: Discrete variables, they take a set value(yes or no, spam or ham, etc.)

Regression model, and example for one feature

Given a set of data, such as the prices of houses by their respective sizes, can we predict the price of house for an specific size?

Lets introduce the concept of training data: Given an data set containing features(inputs, for example house size, the $x's$) and known values for their outputs(for example, price, the $y's$) learn a hypothesis ($h(x)$) that maps h:$x \to y$, where h is a good predictor for the corresponding value of y. Definitions:

1. m = Number of training examples
2. x's inputs/ features (can be more than one)
3. y's = outputs / ”target”, predictions
4. (x,y) = training example
5. ($x^{(i)}$, $y^{(i)}$) = i^{th} training example. Giving the values of features $x^{(i)}$, y^i is the output for the i^{th} example
6. In case of more than one feature, $x^{(i)}$ is a vector, and them $x_j^{(i)}$ is the value of the j-th feature of the i-th training example.
7. n = number of features.

Example(for random values):

Size(feet^2) (x_1)	Price (y)
2104	460
146	700
325	900
412	1215
500	7000

Modeling the function $h(x)$, for example with we want to model this by a linear function, we can try:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 \tag{1.1}$$

So given an unknown input x we can get a good prediction for the output.

Idea: find the set of parameters(θ_0, θ_1) that makes $h_\theta(x^{(i)})$ the closest to the real value $y^{(i)}$!

1. Notes, vectorization for one feature (n=1):
   We can vectorize $h_\theta(x^{(i)})$ by introducing a new dummy variable $x_0^{(i)} = 1$, so:
   $h_\theta(x^{(i)}) = \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} = \sum_{j=0}^{j=1} \theta_j^{(i)} x_j^{(i)}$
   Defining:

   $$\vec{\theta} = \begin{bmatrix}\theta_0 \\ \theta_1\end{bmatrix} \quad (1.2) \quad \vec{x}^{(i)} = \begin{bmatrix}x_0^{(i)} \\ x_1^{(i)}\end{bmatrix} \quad (1.3)$$

   We can write:

   $$h_\theta(x^{(i)}) = \vec{\theta}^T \vec{x}^{(i)} \tag{1.4}$$

2. $h_\theta(x^{(i)})$ is the value of a single prediction for the i^{th} training sample. It is interesting to define $\vec{H}$ and $\vec{y}$ that contains the predictions for all training samples, and the expected value for this prediction:

   $$\vec{H} = \begin{bmatrix} h_\theta(x^{(1)}) \\ h_\theta(x^{(2)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{bmatrix} \quad (1.5) \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \quad (1.6)$$

3. We can build $\vec{H}$ by defining a new matrix $\mathbf{X}$ where is it given in such

way that each row is x^T of the i^{th} training sample:

$$\mathbf{X} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \\ \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} \end{bmatrix} \tag{1.7}$$

So:

$$\vec{H} = \begin{bmatrix} h_\theta(x^{(1)}) \\ h_\theta(x^{(2)}) \\ \vdots \\ h_\theta(x^{(m)}) \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{j=1} \theta_j^{(1)} x_j^{(1)} \\ \sum_{j=0}^{j=1} \theta_j^{(2)} x_j^{(2)} \\ \vdots \\ \sum_{j=0}^{j=1} \theta_j^{(m)} x_j^{(m)} \end{bmatrix} = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \\ \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = X * \theta \tag{1.8}$$

4. Note that the matrix **X** is (m,2). If we generalize for n features, it is simple to see that the matrix is now (m,n+1), and both $x^{(i)}$ and θ becomes ((n+1),1) matrices, but the calculations remains the same!

5. The model is as general as possible, so we can fit any kind of polynomial model as we want. For eg, if we want a quadratic model, we just set x_1 in such way that $x_1 = x^2$, or we can simple combine features and create $x_2 = x_1^2$ and set our function as $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 ...$ etc.

Cost function

The cost function is defined as:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{i=m} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m} (\vec{H} - \vec{y})^T (\vec{H} - \vec{y}) \tag{1.9}$$

This is also known as the mean squared error. This cost functions tell us how good is our estimate for our set of parameters θ, or the accuracy of our

hypothesis function, given the input's x to our hypothesis function, how close it gets to the actual outputs y.
Or in another way, given a hypothesis function, with a set values of θ, how good it is the output given the inputs x's from training set compared to the expected output y of the training set.
So going back to the idea, of finding the best values of θ that gives me the best mapping $h : X \rightarrow Y$, we want to minimize this function!

1. Notes: $\forall \theta_0, \theta_1 etc..$ J $\geqslant 0$
2. With only one feature, it is simple to see then that the minimum is always the root of J, so J=0 (see Course section of intuition)
3. In the perfect case of h=y, J=0.

What we need: Find the values of θ that minimizes $J(\theta_0, \theta_1)$.

1.3 Minimizing $J(\theta)$

We have the cost function $J(\theta)$ that measures the accuracy of our current hypothesis function.
What we want: To find the values of θ that minimizes it. Basic idea:

1. Start the values of θ to a certain number (for eg: for n=1, we set the values of θ_0 and θ_1 to some number)
2. Change this values until it reduces $J(\theta)$
3. Continue changing until (hopefully) we get to a minimum

Gradient Descent

To achieve this we have the gradient descent algorithm given by:
Repeat until converge:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} \tag{1.10}$$

The intuition is nicely given in the course, but just thinking its easy too se that:

1. $\frac{\partial J(\theta)}{\partial \theta_j}$is > 0 if im after the minimum so each step reduces θ in direction of the closest minimum
2. $\frac{\partial J(\theta)}{\partial \theta_j}$ is < 0 if i am before the minimum, so each step increases θ in direction of the minimum!

Taking the derivative of Eq. 1.9, the gradient descent algorithm becomes:

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{1.11}$$

Notes:

1. This is easy to see as $h_\theta(x^{(i)}) = \sum_{j=0}^{j=n} \theta_j x_j^{(i)}$ so $\frac{\partial h_\theta(x^{(i)})}{\partial \theta_j} = x_j^{(i)}$
2. At each step all θ's have to be updated simultaneously!
3. α is the learning rate, or how small or big its the steps i take. If it is too big, it is possible to never converge as i overshoot the minimum and if its too small, the time to converge can be really long.
4. j $\in$ [0,n]

Gradient Descent for Linear Regression

Returning to the simple case of n=1, for a simple linear regression we have:

$$\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \tag{1.12}$$

$$\theta_1 := \theta_1 - \frac{\alpha}{m}\sum_{i=1}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)} \tag{1.13}$$

Remembering that each step we do simultaneous updates! We repeat this process until we get minimize our cost function

Vectorized Gradient descent

We have:

$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \tag{1.14}$$

This can be written as:

$$\theta_j := \theta_j - \frac{\alpha}{m}x_j^T(\vec{H} - \vec{y}) \tag{1.15}$$

where

$$x_j^{(i)} = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ x_j^{(3)} \\ \vdots \\ x_j^{(m)} \end{bmatrix} \tag{1.16}$$

So its easy to see that:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \frac{\alpha}{m}\mathbf{X}^T(\vec{H} - \vec{y}) \tag{1.17}$$

More explicitly:

$$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} - \begin{bmatrix} x_0^{(1)} & x_0^{(2)} & x_0^{(3)} & \dots & x_0^{(m)} \\ x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \dots & x_1^{(m)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^{(1)} & x_n^{(2)} & x_n^{(3)} & \dots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \\ \sum_{i=0}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)} \\ \vdots \\ \sum_{i=0}^{i=m}(h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)} \end{bmatrix} \tag{1.18}$$

Feature Normalization

In we cases when we have many features, they can have different ranges of values:

1. $0 \leq x_1 \leq 1000$
2. $0 \leq x_2 \leq 600000$

3. $\vdots$

The gradient descent algorithm maybe become slow, as θ will descend quickly for smaller ranges and slowly on larger ranges. So we normalize our features, in such way that:

$$x_i := \frac{x_i - \mu_i}{\sigma} \tag{1.19}$$

Where μ_i is the mean value of the feature x_i and σ is the associated standard deviation. Therefore our features will have a nice smaller range like, eg: $-1.5 \leq x_i \leq 1.5$

Normal Equation

It is possible to solve analytically the minimization equation, and get directly the values for θ:

$$\theta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\vec{y} \tag{1.20}$$

However, since we have to calculate the inverse of a matrix, the order of complexity of this algorithm is O(n^3). So comparing with the order of complexity of the gradient descent O(kn^2), as we increase the number of features it can be really slow to compute this normal equation, and the iterative process is a better option.
In contrast, we dont need to chose an appropriate learning rate α, and feature normalization is irrelevant for this case. So we should always try to pick the best option depending on the problem!