The course is available at : [Supervised Machine Learning: Regression and Classification - Week 1: Introduction to Machine Learning - Week 1 | Coursera]

Regression Model

The cost is a measure how well our model is predicting the target. The following formula shows the cost function of linear regression with one variable:

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

Note that we divide m to obtain an average, and further divide 2 for make calculations neater.

The fact that the cost function squares the loss ensures that the 'error surface' is convex like a soup bowl. It will always have a minimum that can be reached by following the gradient in all dimensions.

Gradient Descent

Repeat the following steps until convergence(∂ refers to the partial derivative):

$$w = w - \alpha\frac{\partial}{\partial w}J(w,b)$$

$$b = b - \alpha\frac{\partial}{\partial b}J(w,b)$$

Here are some explanations:

- **Simultaneous update** : you calculate the partial derivatives for all the parameters before updating any of the parameters.

- **Obtain local minimum** : when we reach the local minimum, the derivative will be zero and there's no further movement.
- **Appropriate learning rate** : small alpha may be slow, while large alpha result in overshooting.
- **Fixed learning rate** : the length of a single step varies with the partial derivative. So we can reach the local minimum with a fixed learning rate.

Let's simplify the formula above and see what we get(using partial derivative rules):

$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)}$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

How to solve the gradient descent with multiple variables. Actually, it is as much same as the linear regression.

$$w_j = w_j - \alpha \frac{\partial J(\mathrm{w},b)}{\partial w_j}$$

$$b = b - \alpha \frac{\partial J(\mathrm{w},b)}{\partial b}$$

where, n is the number of features, parameters w_j, b are updated simultaneously and where:

$$\frac{\partial J(\mathrm{w},b)}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) \times x_j^{(i)}$$

$$\frac{\partial J(\mathrm{w},b)}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

The implement of the gradient is under the file `graident descent`. You can check the code as you want.

Here are some advice when debugging the algorithm :

- For sufficiently small α, $J(w,b)$ should decrease on every iteration.
- Convergence test : if $|\Delta J(w,b)| \leq 10^{-3}$, we declare that the model reaches convergence.

Feature Scaling

Feature scaling refers to rescaling the dataset so the features have a similar range.

Assume μ_j is the mean of all the values for feature (j), σ_j is the standard deviation of feature (j). To implement z-score normalization, adjust your input values as shown in this formula:

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

> Implementation Note : Not only should we rescale the training data, but also the testing ones.

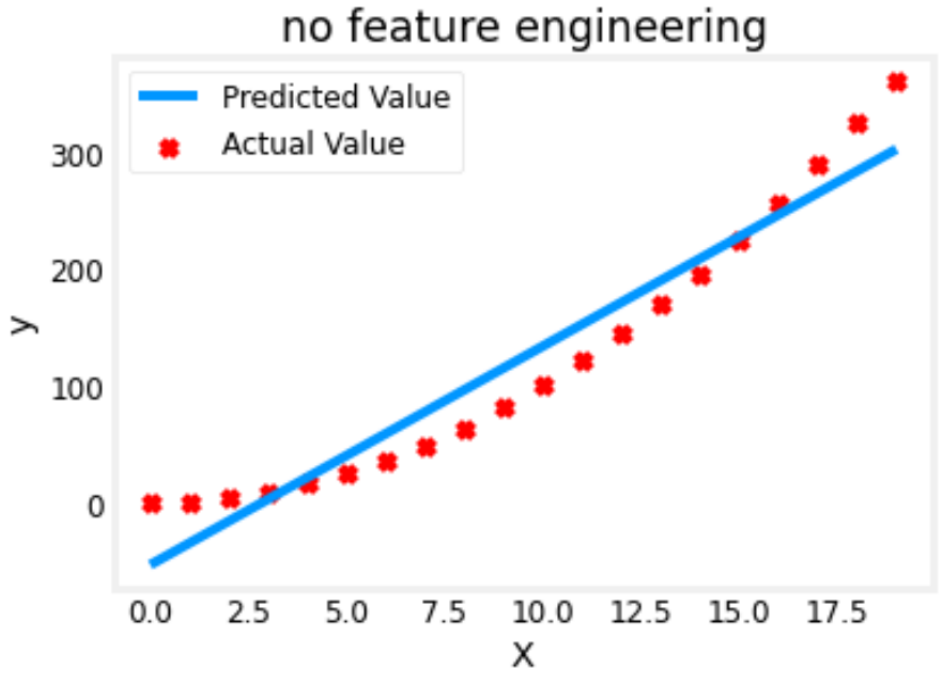
Still, we have another normalization approach called Min-Max normalization :

$$x_j^{(i)} = \frac{x_j^{(i)} - \min}{\max - \min}$$

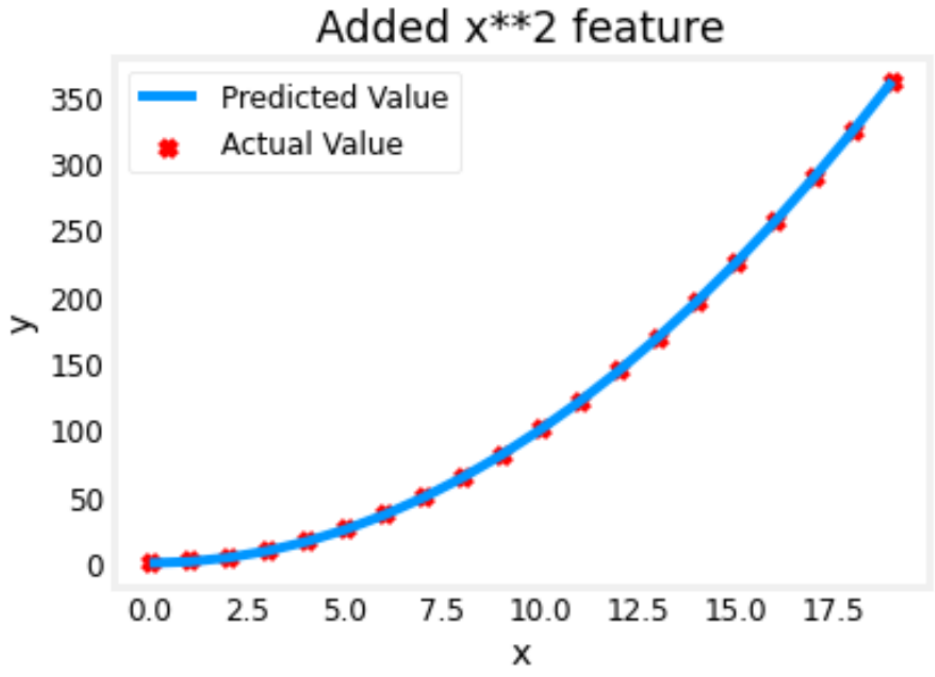
Feature Engineering

Feature engineering refers to using domain knowledge to design new features by transforming a feature or combining features.

We'll start with a simple quadratic: $y = 1 + x^2$:



The graph suggests the linear regression is not a great fit. What we need is something like $y = wx^2 + b$, or a polynomial feature. To accomplish this, we can replace x with x^2 :



Near perfect fit! Although we know that an x^2 term was required, it may not always be obvious which features are required. One could add a variety of potential features to try and find the most useful. For example, if we get the hypothesis:

$0.08x + 0.54x^2 + 0.03x^3 + 0.0106$

Gradient descent is picking the "correct" features for us by emphasizing its associated parameter, since the weight associated with x^2 feature is much larger than the weight associated with x or x^3 feature.

If the data set has features with significantly different scales, one should apply feature scaling to speed gradient descent. So we apply feature scaling to the polynomial features.

Logistic regression

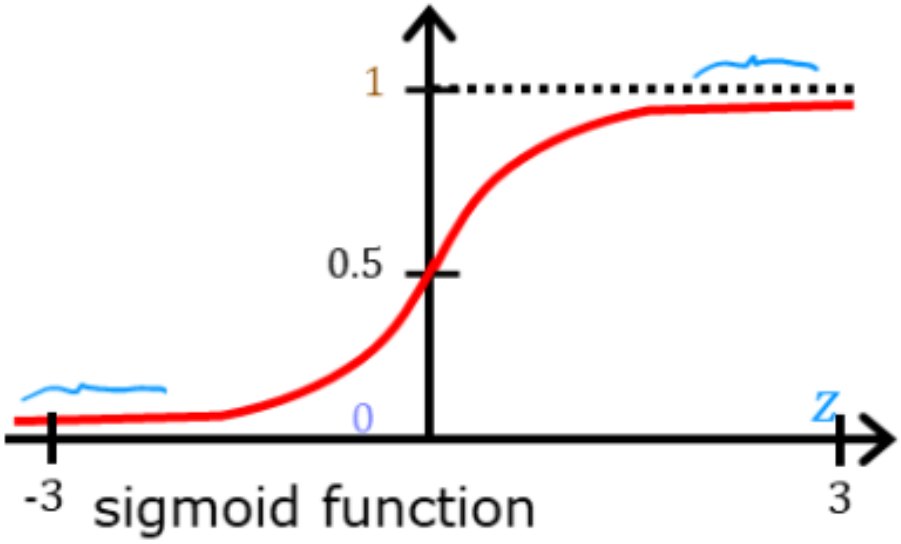
Let's discuss a classic problem: binary classification.

It turns out that the linear regression is not suitable for classification problem. However, we would like the predictions of our classification model to be between 0 and 1 since our output variable y is either 0 or 1.

This can be accomplished by using a "sigmoid function" (or logistic function) :

$f(x^{(i)}) = \frac{1}{1+e^{-(x^{(i)}\vec{w}+b)}}$

When predicting, if $f(x^{(i)}) \geq 0.5$, we predict the output is 1. Otherwise we predict 0.



Now let's dive into the loss function and the cost function. Note that loss is a measure of the difference of a single example to its target value while the cost is a measure of the losses over the training set.

If we apply the squared error cost function into the logistic regression, it turns out that the cost function will be **non-convex**, which is not appropriate for gradient descent. We need a new loss function:

$$loss(i) = \begin{cases} -\log(f(x^{(i)})) & y^{(i)} = 1 \\ -\log(1 - f(x^{(i)})) & y^{(i)} = 0 \end{cases}$$

The loss function above can be rewritten to be easier to implement:

$$loss(i) = -y^{(i)} \log(f(x^{(i)})) - (1 - y^{(i)}) \log(1 - f(x^{(i)}))$$

$$J(w, b) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(f(x^{(i)})) + (1 - y^{(i)}) \log(1 - f(x^{(i)}))]$$

Recall the gradient descent algorithm utilizes the gradient calculation:

$$w_j = w_j - \alpha \frac{\partial J(w,b)}{\partial w_j}$$

$$b = b - \alpha \frac{\partial J(w,b)}{\partial b}$$

Let's get down to the $\frac{\partial J(w,b)}{\partial w_j}$ and see what we can get :

$$\begin{aligned}\frac{\partial J(w,b)}{\partial w_j} &= -\frac{1}{m}\sum_{i=1}^{m}\frac{y}{f(x)} - \frac{1-y}{1-f(x)} \cdot f'(x) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\frac{y-f(x)}{f(x)(1-f(x))} \cdot f'(x)\end{aligned}$$

There is a quite interesting conclusion. For $g(z) = \frac{1}{1+e^{-z}}$, we have $g'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = g(z)(1-g(z))$. Using chain rule, we can rewritten the expression into the following form:

$$\begin{aligned}\frac{\partial J(w,b)}{\partial w_j} &= -\frac{1}{m}\sum_{i=1}^{m}\frac{y-f(x)}{f(x)(1-f(x))} \cdot f(x)(1-f(x)) \cdot x_j \\ &= \frac{1}{m}\sum_{i=1}^{m}(f(x)-y) \cdot x_j\end{aligned}$$

Ultimately, the gradient calculation looks like this:

$w_j = w_j - \frac{1}{m}\sum_{i=1}^{m}(f(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$

$b = b - \frac{1}{m}\sum_{i=1}^{m}(f(x^{(i)}) - y^{(i)})$

That's unbelievable! The gradient descent is completely the same as the one in linear regression. Maybe the only difference is the definition of loss function. But it's still amazing!

Regularization

I would like you to think about how to address overfitting. Here are some useful advice:

- Collect more data.
- Select features (or more Specifically, choose a subset)

- Reduce the size of parameters —— **Regularization**

Regularization discourages large weights for particular feature. Since we know that simpler model is less likely to overfit, let's penalize all of the weights a bit and shrink them by modifying the cost function:

$$J(\vec{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (f_{\vec{w},b}(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m} \sum_{i=1}^{n} w_j^2$$

When implementing gradient descent, we just need to change the expression a bit:

$$w_j = w_j - \alpha \frac{\lambda}{m} w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (f(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

From a mathematical perspective, the effect of this term is that on every single iteration of gradient descent, we are multiplying the w_j by a number sightly less than 1.

What I mentioned above explain how regularization works in linear regression. It is as much same as the one in logistic regression. Regularization promotes model simplicity and stability in order to avoid overfitting.