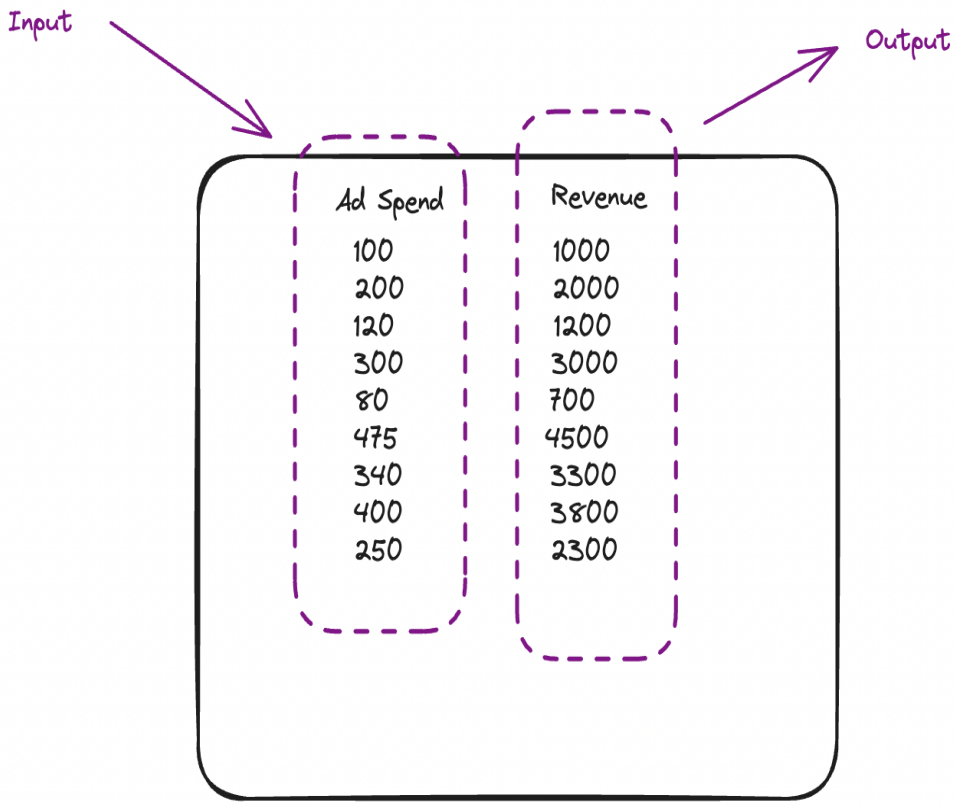
Linear Regression

Before reading this blog post, please read the post on Introduction to Machine Learning.

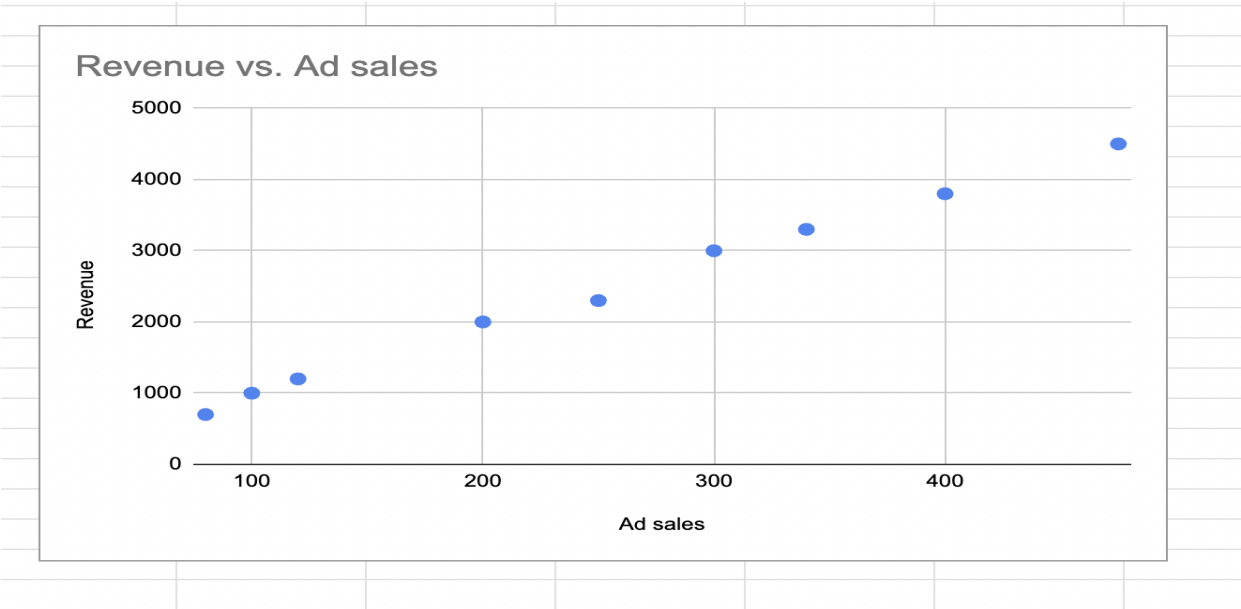
Straight line with minimal error

We will look at the linear regression algorithm, which is one of the oldest and popular supervised learning algorithms. In supervised learning, we are trying to approximate the given input and output data with a function which represents its relationship. In the case of linear regression, the function is going to be a **straight line**.

We will look at the example of an online company predicting their revenue from the ad spend based on historical data.

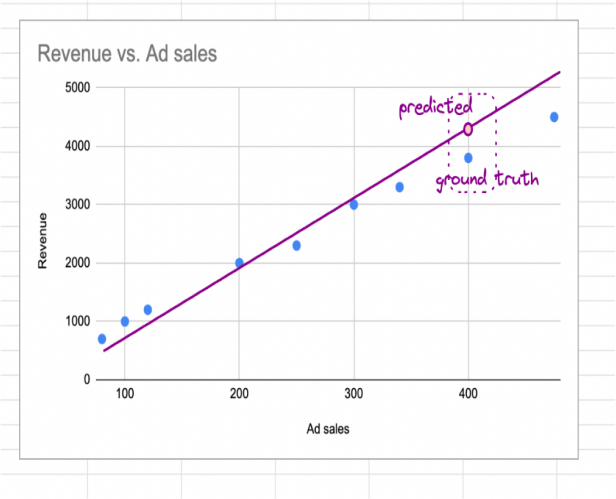


If we graph the above data using a scatter plot, we get,



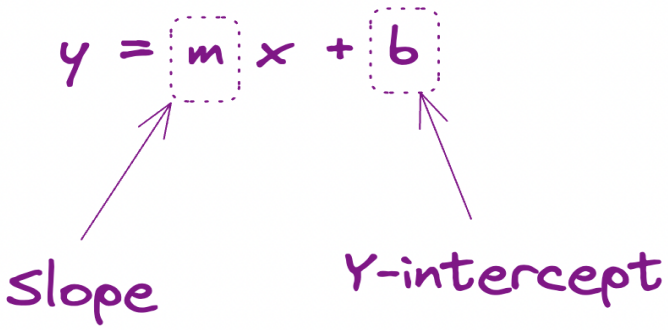
As I mentioned earlier, in the case of linear regression, we use a straight line function to approximate the data points. But we can draw infinite straight lines in the above chart. So, how do we pick 1 straight line which does the best approximation of the data points from the infinite number of straight lines which can be possibly drawn?

What if we pick a line, which gives the lowest error between the given output values and the predicted output values by our straight line.



Equation of a straight line

While studying mathematics in school, we would have studied linear equations which provides the equation for a straight line. The most common form is the slope-intercept equation of a straight line:



where m is the slope (or) average rate of change. To compute average rate of change (or) slope between 2 points, we need to compute rise over run (change in y over change in x):

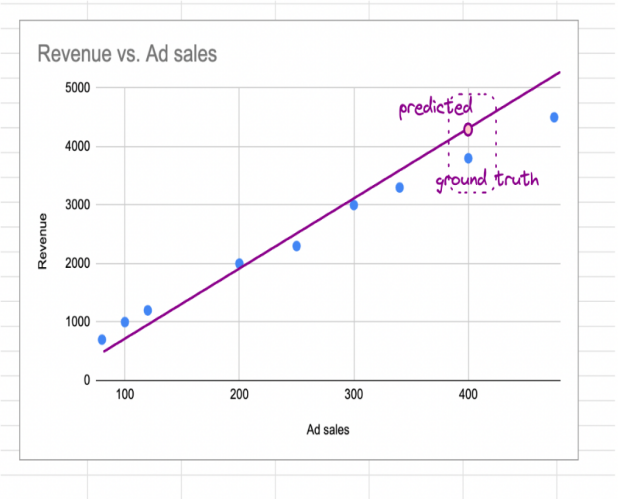
$$\textbf{Slope between 2 points} = y_2 - y_1 / x_2 - x_1$$

and **b** is the **y-intercept**, which is the value of y where the line crosses the y-axis. All we need to draw a straight line is m and b.

In the parlance of machine learning, slope is called **weight**, y-intercept is called **bias**. Using a linear regression algorithm, we need to find appropriate weight and bias, for which we have minimal error between the predicted value and ground truth.

Mean Square Error function

We need to find the weight and bias for a straight line which gives the lowest error between the given output value and the predicted output value.



For 1 example, we can compute the error between predicted output and ground truth as:

Error = Predicted value - Ground truth
Error = $(w_0 + w_1 x_1)$ - Ground truth

Since we will have m examples, we need to compute the error for all the examples. If some errors are +ve and some errors are -ve, we don't want them to cancel each other out. So we would want to square the errors.

$(\text{Error})^2 = ((w_0 + w_1 x_1) - \text{Ground truth})^2$

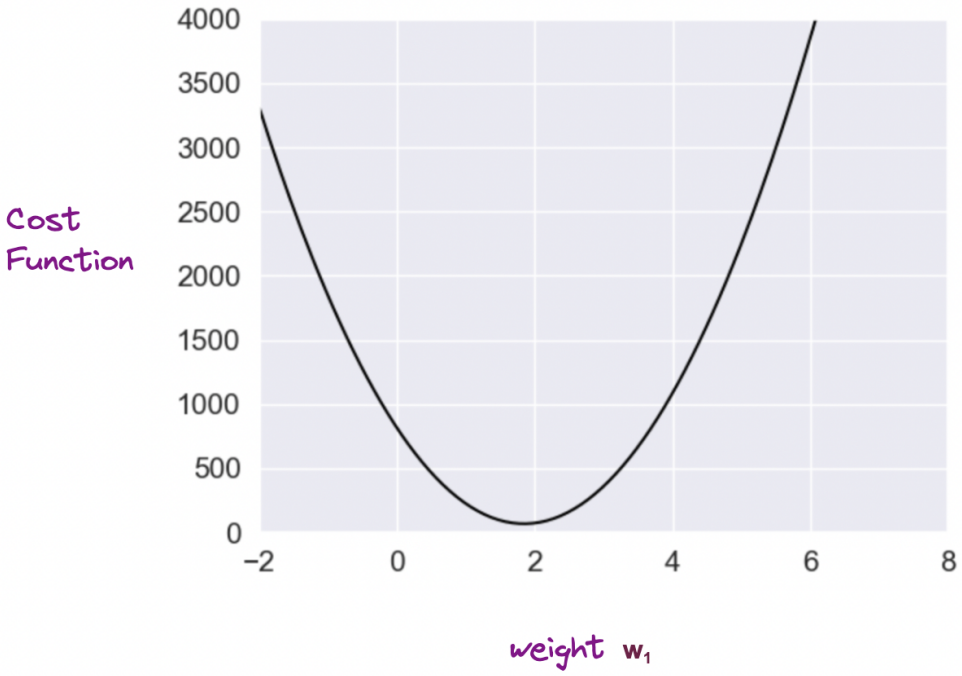
Average Error for m examples is called the Cost function which is called as Mean Squared Error:

Cost Function (Mean Squared Error) = $\sum_{i=1}^{n}$ **$((w_0 + w_1 x_i) - \text{Ground truth})^2$ / n**

Our optimization function needs to figure out w_0, w_1 where the Cost function is minimum.

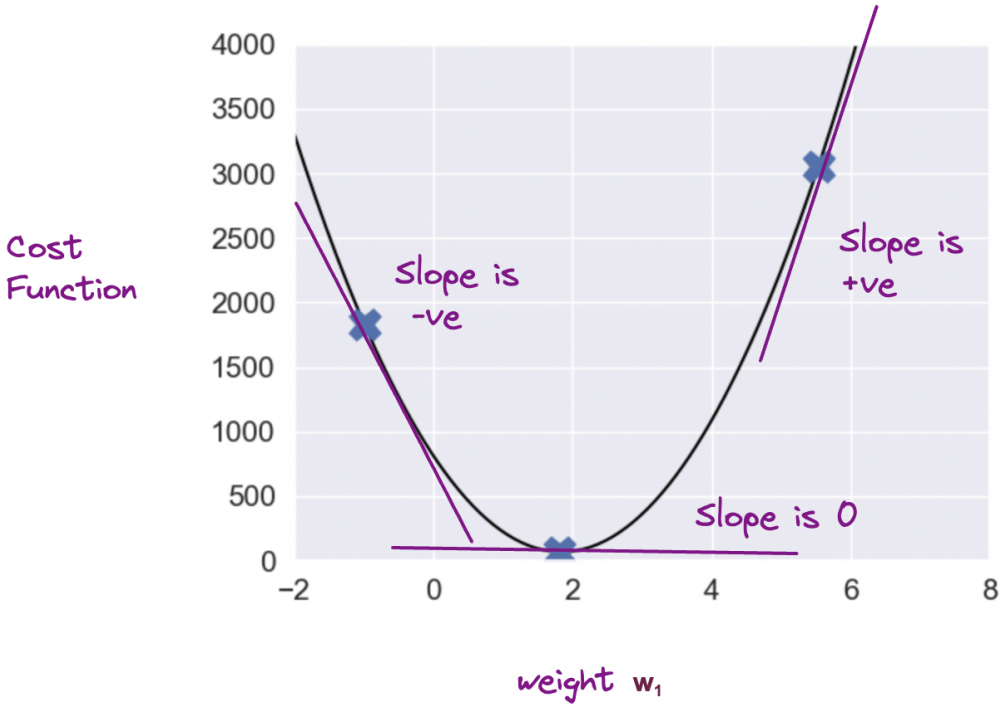
Gradient descent of Mean Square Error function

Right now, the cost function has 2 independent variables w_0 , w_1 and cost function being the dependent variable. To be able to visualize in a cartesian plot, we will ignore w_0. We can see that the relationship between cost function and w_1 is **quadratic** in nature.



Assume that we pick a random weight, say w_1 = -1.0. How do we come up with machine learning algorithm that should move to the right of w_1 at -1.0 all the way closer to 2 and stop there, so that it yields the value of w_1 at which the cost function is at the minimum?

In Calculus, for functions which are **continuous** and **differentiable**, we can compute derivatives which gives us the instantaneous rate of change of a function. Derivatives for a function f(x) at a given point gives us 2 things: **magnitude** and **direction**. By using the **magnitude** and **direction** from the derivative value, we can have the algorithm take the step in the right direction with **precision** and **speed**, which will lead to a point where the cost function is minimum. On top of functions being **continuous** and **differentiable** we would also want the function to be **convex** so that if a minimum exists, it will be the global minimum. This algorithm is called **gradient descent**. To learn more about Key Ideas of Calculus, please go here.



Now that we understand the gradient descent algorithm, we will also fix the w_0 (bias) parameter. Our line equation was $w_0 + w_1 x_i$, and now the cost function is not a two dimensional curve. From a line it has become a surface. Fortunately, we can still use the gradient descent algorithm for a function with multiple variables by applying partial derivatives. Using partial derivatives, you take each variable and assume that it is the only variable in the cost function and all other variables are constant. So, when we take the partial derivative of cost function w.r.t w_1, then w_0 is constant and similarly when we take the partial derivative of cost function w.r.t w_0, then w_1 is constant

Cost Function or J(w) = $\left(\sum_{i=1}^{n}\right)$ **$((w_0 + w_1 x_i)$ - Ground truth$)^2$) / n**

Partial Derivative of $J(w_1)$ w.r.t w_1 = (2 $\sum_{i=1}^{n}$ **$((w_0 + w_1 x_{1(i)}))$ * $x_{1(i)}$) / n**

Partial Derivative of $J(w_0)$ w.r.t w_0 = (2 $\sum_{i=1}^{n}$ **$((w_0 + w_1 x_{1(i)}))$) / n**

If we take closer attention between the partial derivative of cost function w.r.t w_1 and partial derivative of cost function w.r.t w_0 the only difference is $x_{0(i)}$ is missing in the derivative of cost function w.r.t bias function. For bias, If we treat $x_{0(i)} = 1$ as a special case, then there is no difference between the two partial derivatives.

In our current example, we had one feature “Ad Spend” but in real world machine learning problems, we will end up having many thousands of features. To solve this, we will be using **multiple linear regression**.

Our function will be, $w_0 x_0 + w_1 x_i + w_2 x_2 + w_3 x_3 \ldots + w_m x_m$ where w_0 is for bias and $x_0 = 1$.

Cost Function = $\left(\sum_{i=1}^{n} ((w_0 x_{0(i)} + w_1 x_{1(i)} + w_2 x_{2(i)} + \ldots + w_m x_{m(i)}) - \text{Ground truth})^2\right) / n$

Partial Derivative = $\left(2 \sum_{i=1}^{n} ((w_0 x_{0(i} + w_1 x_{1(i)} + \ldots + w_m x_{m(i)}) - \text{Ground truth}) * x_{m(i)}\right) / n$

($J(w_m)$ w.r.t w_m)

The derivative can be worked out by knowing the derivatives of a few key functions and applying the **“Chain rule”** of Calculus.

Solution

To solve multiple linear regression, all we have to do is compute the partial derivatives for our cost function w.r.t each of the features and using the **magnitude** and **direction** of their derivative value, we will adjust all the feature’s weights accordingly so that we find the appropriate weights where the cost function is minimum. The adjustment is made using **learning rate**, which will be a small value like 0.1, 0.01, 0.001 etc.,

```
def predict(X, w):
    return np.matmul(X, w)


def gradient(X, Y, w):
    return 2 * np.matmul(X.T, (predict(X, w) - Y)) / X.shape[0]


def train(X, Y, iterations, lr):
    w = np.zeros((X.shape[1], 1))
    for i in range(iterations):
        w -= gradient(X, Y, w) * lr
    return w
```

Partial Derivative = (2 $\sum_{i=1}^{n}$ (($w_0 x_{0(i} + w_1 x_{1(i)} + ... + w_m x_{m(i)}$) - Ground truth) * $x_{m(i)}$) / n

($J(w_m)$ w.r.t w_m)

(Note: we compute partial derivative of cost function w.r.t each of the weights)

To solve the partial derivative, we use matrix operations like matrix multiplications and matrix transpose. In the case of multiple linear regression, the model equation is given as $w_0 x_0 + w_1 x_i + w_2 x_2 + w_3 x_3 ..... + w_m x_m$ where w_0 is for bias and $x_0 = 1$ where,

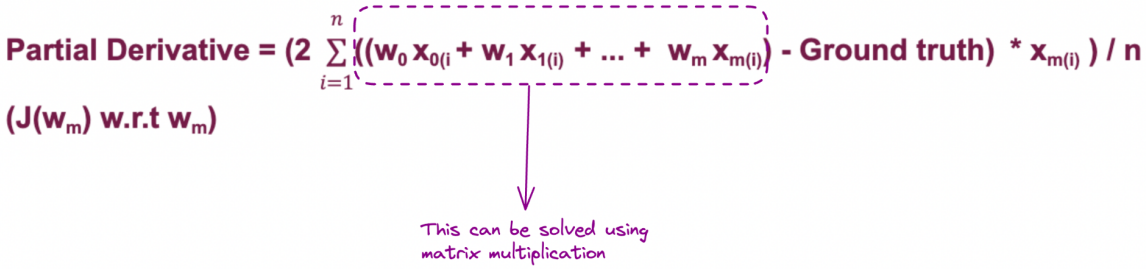
w_0 w_1 ..... w_m - Weights of the features
x_0 x_1 ..... x_m - Features
y^{prime} - Predicted output

If we have n examples, 1 output per example and m features, we can represent the model equation using a matrix multiplication:

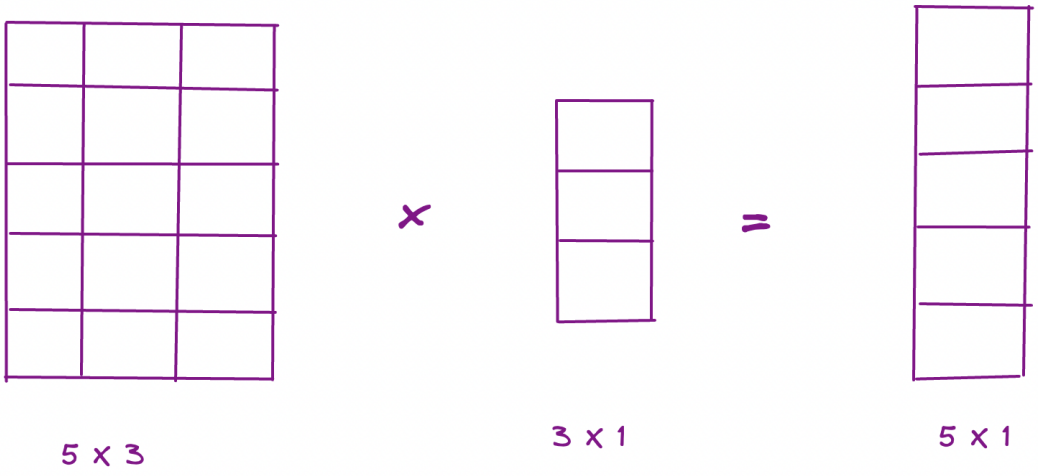
Examples X Features (n X m)	**X**	Features X Output (m X 1)	=	Examples X Output (n X 1)

where **X** represents matrix multiplication

We can use matrix multiplication as shown below while we are computing partial derivatives.

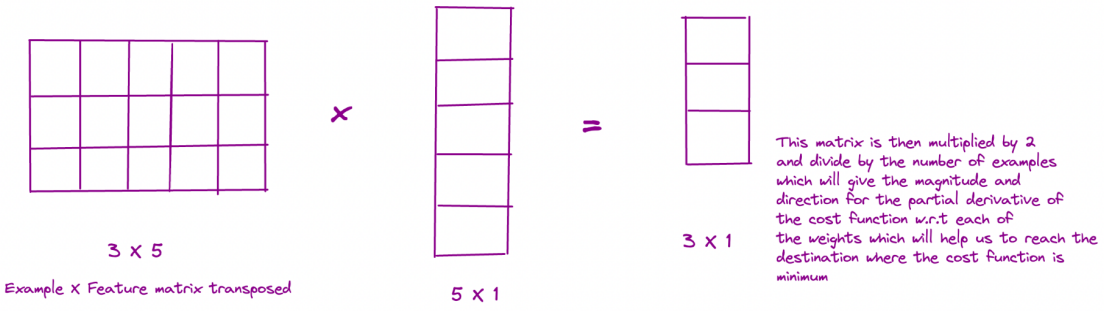


For ex, if we have 5 examples, 3 features(including bias), 1 output, matrix multiplication will be,



y^{prime} - **Ground Truth,** where y^{prime} is predicted value and the result will also be a **5 x 1** matrix

For the partial derivatives, now we need to multiply $x_{m(i)}$ with y^{prime} - **Ground Truth**. This is where we need to transpose the example x features of (n x m) to (m x n). In our example, we will be transposing our (5 x3) matrix to (3 x 5) matrix. The output is then multiplied by 2 and divided by the number of examples which will give the magnitude and direction for the partial derivative of the cost function w.r.t each of the weights which will help us to reach the destination where the cost function is minimum.



We will adjust all the weights together and will perform this step for i iterations. At the end of the iterations, the adjusted weights are considered to be the ones, where the cost function is minimum.

Conclusion

In this post, we covered Linear regression which is one of the most popular supervised learning models. Next up, we will look into another popular supervised learning model, which is Logistic Regression, which is used when we need to categorize a set of things.

References

Programming Machine Learning : From Coding to Deep Learning
Linear Regression