# Chapter 4 : Training Models

## Linear Regression :

A linear model makes predictions by simply computing a weighted sum of the input features, plus a constant called the bias term. (Also called the intercept term)

Eqn 4-1 Linear Regression model prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots \theta_n x_n$$

where

$\hat{y}$ is the predicted value

$n$ is the number of features

$x_i$ is the $i^{th}$ feature value

$\theta_j$ is the $j^{th}$ model parameter (including the bias term $\theta_0$ and the feature weights $\theta_1, \theta_2, \dots, \theta_n$)

This can be written more concisely in vectorized form.

Eqn 4-2 Lin. Reg. model prediction vectorized form

$$\hat{y} = h_{\vec{\theta}}(\vec{x}) = \vec{\theta} \cdot \vec{x}$$

where

$\vec{\theta}$ is the model's parameter vector, containing the bias term $\theta_0$ and feature weights $\theta_1 \dots \theta_n$

$\vec{x}$ is the instance's feature vector, containing $x_0$ to $x_n$, with $x_0$ always equal to 1

$\vec{\theta} \cdot \vec{x}$ is the dot product of vectors $\vec{\theta}$ and $\vec{x}$ which is equal to $\theta_0 x_0 + \theta_1 x_1 + \dots \theta_n x_n$

$h_\theta$ is the hypothesis function, using the model parameters $\vec{\theta}$

How do we train our Lin. Reg. Model?

To train it we need to find the value of $\vec{\theta}$ that minimizes the RMSE. In practice, it is easier to minimize the MSE

The MSE of a Lin. Reg. hypothesis $h_\theta$ on a training set $\vec{X}$ is calculated using Eqn. 4-3

Eqn 4-3 MSE cost function for a Lin. Reg. model

$$MSE(\vec{X}, h_\theta) = \frac{1}{m} \sum_{i=1}^{m} (\vec{\theta}^T \vec{x}^{(i)} - y^{(i)})^2$$

To simplify notations we will just write $MSE(\vec{\theta})$

## Normal Equation:

We can find the value of $\vec{\theta}$ that minimizes the cost function directly by using the closed-form solution called the Normal Equation

Eqn 4-4 Normal Equation

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

where

$\hat{\vec{\theta}}$ is the value of $\vec{\theta}$ that minimizes the cost function

$\vec{y}$ is the vector of target values containing $y^{(1)}$ to $y^{(m)}$

Let's test this in our jupyter notebook

Note that the Normal Equation may not work if the matrix $\vec{X}^T \vec{X}$ is not invertible (i.e. singular) such as if $m < n$ or some features are redundant.

## Computational Complexity:

The Normal Equation computes the inverse of $\vec{X}^T \vec{X}$ which is an $(n+1) \times (n+1)$ matrix.

The computational complexity of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$.

Thus, if you double the number of features ($n$) you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$. The SVD (Singular Value Decomposition) approach used by Scikit-Learn's LinearRegression class is $O(n^2)$, so if you double the number of features you multiply the computation time by $2^2 = 4$.

Both the Normal Equation and SVD approach get very slow when the number of features grows large (ex. 100,000) but on the positive side, both are linear with regards to the number of instances in the training set ($O(m)$), so they handle large training sets efficiently provided they can fit in memory.

## Gradient Descent:

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of G.D. is to tweak parameters iteratively in order to minimize a cost function.

You start by filling $\vec{\theta}$ with random values. This is called random initialization.

An important parameter in G.D. is the step size determined by the 'learning rate' hyperparameter.

The MSE cost function for a linear regression model is a convex function which implies there are no local minima, just one global minimum.

When using G.D. you should ensure all features have a similar scale.

## Batch Gradient Descent:

To implement G.D. we need to compute the gradient of the cost function with regard to each model parameter $\theta_j$.

Eqn. 4-5 Partial Derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} MSE(\vec{\theta}) = \frac{2}{m} \sum_{i=1}^{m} (\vec{\theta}^T \vec{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Instead of computing each partial derivative individually, we can use eqn. 4-6 to compute them all in one go.

The gradient vector $\nabla_\theta MSE(\vec{\theta})$ contains all the partial derivatives of the cost function. (one for each model parameter)

Eqn. 4-6 Gradient vector of the cost function

$$\nabla_\theta MSE(\vec{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\vec{\theta}) \\ \frac{\partial}{\partial \theta_1} MSE(\vec{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} MSE(\vec{\theta}) \end{pmatrix} = \frac{2}{m} \vec{X}^T (\vec{X}\vec{\theta} - \vec{y})$$

Notice that this calculates over the full training set $\vec{X}$ at each G.D. step.

Batch Gradient Descent == Uses the whole batch every step.

Eqn. 4-7 Gradient Descent Step

$$\vec{\theta}^{(\text{next step})} = \vec{\theta} - \eta \nabla_{\vec{\theta}} MSE(\vec{\theta})$$

where $\eta$ is the learning rate. (eta == $\eta$)

To find a good learning rate we can use GridSearch
You should limit the number of iterations to eliminate models that take too long to converge.

How to set the number of iterations?
A simple solution is to set a large number of iterations but then to interrupt the algorithm when the gradient vector becomes tiny. That is, when its norm becomes smaller than a tiny number $\epsilon$ called the tolerance; this happens when G.D. has (almost) reached the minimum.

## Stochastic Gradient Descent:

SGD picks a random instance in the training set at every step and computes the gradients based on that single instance. ⇒ Much faster algorithm, Allows larger datasets

⇒ Also a lot more erratic & will bounce around the minimum.

SGD has a better chance of finding the global minimum than BGD.

Randomness (erratic) is good to escape local optima but bad at settling down in the minimum.

One solution is to gradually reduce the learning rate $\eta$. The function that determines the learning rate at each iteration is called the 'learning schedule'.

By convention SGD iterates by rounds of $m$ iterations called 'epochs'

When using SGD the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average.

→ i.e. shuffle your training set properly.

## Mini-batch Gradient Descent:

Mini-batch Gradient Descent computes the gradients on a small random set of instances called mini-batches.