A report on ***Graph learning and its applications***

Sonajhoori Maitra
Under the guidance of:
Professor Sudebkumar Prasant Pal

August 9, 2023

Contents

1 Supervised and Unsupervised Learning- A Basic Understanding **2**

2 Concepts and Tools we will require for understanding Graph Learning **3**

2.1 Representation Learning 3
2.2 Maximum Likelihood Estimation 7
2.3 Softmax function 10
2.4 A Brief Idea of Gradient Descent 12
2.5 Neural Networks - An Overview 14

3 Machine Learning on Graphs - Feature Engineering **30**

3.1 Node Features - Node Degree 32
3.2 Node Features - Node Centrality 32
3.3 Node Features - Clustering Coefficient 37
3.4 Node Features - Graphlets 38

4 An overview of the Node2Vec Paper by Jure Lekstovec and Aditya Grover **38**

1 Supervised and Unsupervised Learning- A Basic Understanding

Supervised Learning is a paradigm of Machine Learning where in addition to data, we also have some kind of supervision associated with the data. In this type of learning, we have prior knowledge of what the output values for our samples should be. Therefore, the goal of supervised learning is to learn a function that, given a sample of data and desired outputs, best approximates the relationship between input and output observable in the data. A simple example would be differentiating between spam and non-spam emails.

Unsupervised Learning, on the other hand, does not have labeled outputs, so its goal is to infer the natural structure present within a set of data points. Here a bunch of data set or data points are given without any labeling and we are expected to make sense from that.

The main distinction between the two approaches is the use of labeled data sets. To put it simply, supervised learning uses labeled input and output data, while an unsupervised learning algorithm does not. In short:

Supervised Learning: The goal is to predict a target output given an input data point.
Unsupervised Learning: The goal is to infer patterns, such as clusters of points, in the data.

- Supervised Learning
  a. Classification
  b. Regression
  c. Ranking
  d. Structure Learning
- Unsupervised Learning
  a. Clustering
  b. Representation Learning

2 Concepts and Tools we will require for understanding Graph Learning

2.1 Representation Learning

Representation Learning is a subcategory of unsupervised learning.

GOAL: Given a set of *data points* , *understand* something *useful* about them.

It is vague and broad because of three different things. One is, we have to define what we mean by "given a set of data points", which leads us to the question: **what are data points?** More importantly, we need to understand, what **understand** and **usefulness** mean. In unsupervised learning, we are just given a bunch of data points. However,we are not given any supervision,

Let us understand what data points are.

We are going to think of **data points** as some vectors in d-dimension. So typically, d real numbers, which we'll call features. For example, if height, weight, and age from 100 different people are collected, each person becomes a three-dimensional point, where each of the coordinates correspond to height, weight and age, in that particular order.

The next question that arises : What does it mean to *understand* something and what does it mean to *understand something useful*?

Comprehension is Compression as quoted by Goerge Catilin answers the above stated questions. Our main motive will be compression of these data points.

Let us understand this with a basic example:
$\{\begin{bmatrix}7\\14\end{bmatrix}, \begin{bmatrix}5\\10\end{bmatrix}, \begin{bmatrix}9\\18\end{bmatrix}, \begin{bmatrix}20\\40\end{bmatrix}\}$ are four data points with two features each.
Naively, there are four data points with two features each so we will need 4*2 = 8 numbers to store this data set. However if we choose a representative vector , say , $\begin{bmatrix}1\\2\end{bmatrix}$, we can represent the data points as $\{\begin{bmatrix}1\\2\end{bmatrix} \begin{bmatrix}7 & 5 & 9 & 20\end{bmatrix}\}$. Now, we need just 6 numbers to represent the data points, resulting in a compression. Though the compression may not seem very significant for just four data points, it is actually pretty significant for a large number of data points.

Let's say we have 1 billion 2- dimensional data points. Naively we will require 2 billion numbers to store these data points. However, if we compress these data points using representatives and coefficients, then we require two numbers for storing the representative and 1 billion numbers for storing the coefficients which give us a total of one billion and two numbers. As we can clearly observe, there is almost a 50% compression.

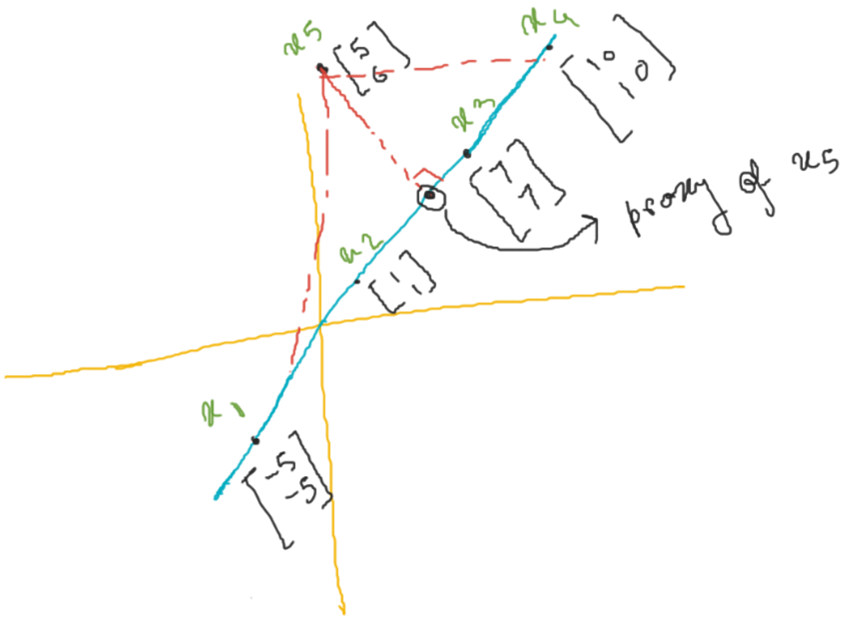
To generalize, for n number of d-dimensional data points, naively we will require nd numbers to store them. However, if we compress these data points using representatives and coefficients, we will require $n+d$ numbers to store these data points.

The data points considered above lie on the line $y = 2x$.

NOTE: Any vector along the line can be chosen as a representative except $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

We took a very trivial example above where all the data points lie on the line $y = 2x$. However, that might not always be the case. In fact, most of the times data points are scattered and do not fall on a single specific line.

Let us understand how we shall optimize for a case where all data points lie on a straight line except one data point. Then we may generalize for more number of data points.



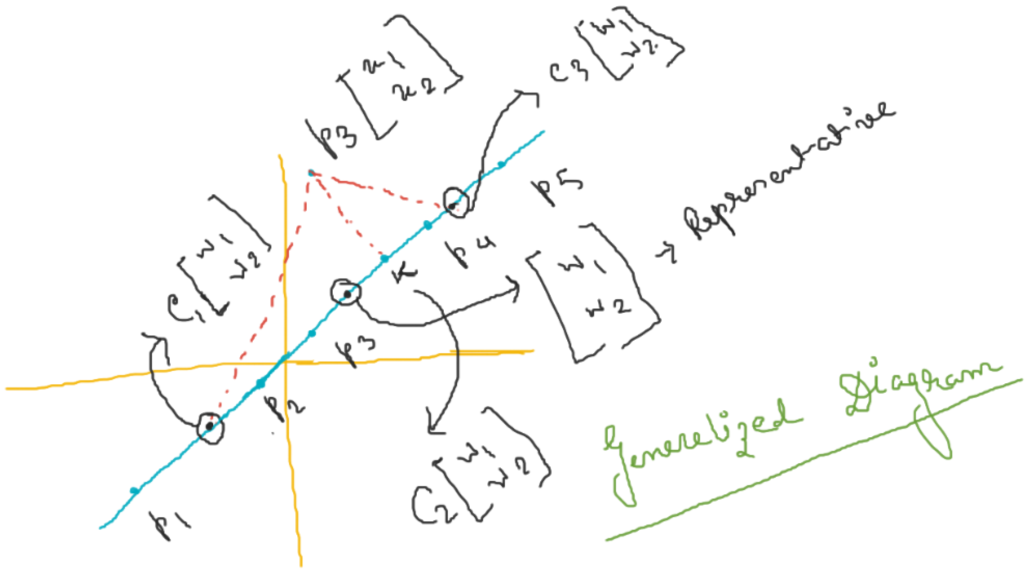
From the above diagram, we can see that, all the data points except $\begin{bmatrix} 5 \\ 6 \end{bmatrix}$, lie on the line $x = y$.

Now, we need to find a *proxy* for $\begin{bmatrix}5\\6\end{bmatrix}$ that lies on the line $x = y$. However, there are infinite number of points on a straight line. So the most important question that arises is:

How do we choose the *proxy* for a data point not lying on the line on which all other or most of the data points are lying?

The answer is: We will consider the projection of the point, say $\begin{bmatrix}x_1\\x_2\end{bmatrix}$ which does not lie on the line , say , $y = mx + c$, onto $y = mx + c$.

It is well implied that we would want to choose the *proxy* in such a way on $y = mx + c$, that it is very close to the point $\begin{bmatrix}x_1\\x_2\end{bmatrix}$. It will result in the least reconstruction error.



Let us consider the Generalized Diagram. we choose a representative $\begin{bmatrix}w_1\\w_2\end{bmatrix}$ that lies on the line. Thus, all points lying on the line can be expressed as c$\begin{bmatrix}w_1\\w_2\end{bmatrix}$, where c is a constant. Similarly, k, the projection of $\begin{bmatrix}x_1\\x_2\end{bmatrix}$ on the line, is also some c$\begin{bmatrix}x_1\\x_2\end{bmatrix}$.

The next question we will deal with is: **What is the c for which the length of the error is as small as possible?**

We can now set this as an optimization problem where we want to minimize the length or the square of length of the error vector with respect to c

The error vector is: $\begin{bmatrix} x_1 - cw_1 \\ x_2 - cw_2 \end{bmatrix}$

$$\begin{aligned} &\min_{\mathbf{c}} \,(\text{length}^2(errorvector)) \\ = &\min_{\mathbf{c}} \,((\text{x}_1 - cw_1)^2 + (x_2 - cw_2)^2 \;) \end{aligned}$$

We know that for finding out the minimum value of c , we can take the first derivative of the function with respect to c and compute it with 0.

$$\begin{aligned} &\tfrac{d((x_1-cw_1)^2+(x_2-cw_2)^2)}{dc} = 0 \\ \implies &\tfrac{d(x_1^2+c^2w_1^2-2x_1cw_1+x_2^2+c^2w_2^2-2x_2cw_2)}{dc} = 0 \\ \implies &2cw_1^2 - 2x_1w_1 + 2cw_2^2 - 2x_2w_2 = 0 \\ \implies &cw_1^2 - x_1w_1 + cw_2^2 - x_2w_2 = 0 \\ \implies &c(w_1^2 + w_2^2) - x_1w_1 - x_2w_2 = 0 \\ \implies &c(w_1^2 + w_2^2) = x_1w_1 + x_2w_2 \\ \implies &c = \tfrac{(x_1w_1+x_2w_2)}{(w_1^2+w_2^2)} \end{aligned}$$

So, from the above calculation we can see that the minimum value of c , let us call it c^*, is : $\frac{(x_1w_1+x_2w_2)}{(w_1^2+w_2^2)}$

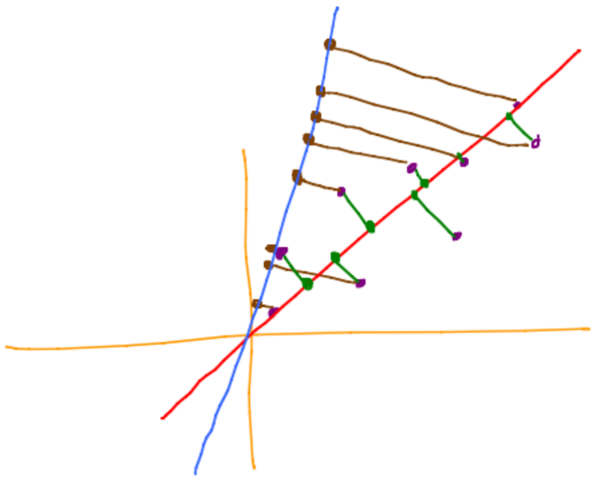
$$\therefore c^* = \tfrac{(x_1w_1+x_2w_2)}{(w_1^2+w_2^2)}$$

On careful observation, we can see that $(x_1w_1 + x_2w_2)$ is the dot product of $\vec{x}$ and $\vec{w}$ and $(w_1^2 + w_2^2)$ is the square of length of $\vec{w}$. Also, we can always pick a representative $\vec{w}$ such that it's length is 1. In that case $c^* = x^Tw$ (Dot product of $\vec{x}$ and $\vec{w}$). Then, the *proxy* point we were looking for will be $c^* \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, i.e, (x^T w) \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$.

Now that we have understood how to find the *proxy* for a single point not lying on a particular line, let us formulate our goal.

GOAL: Develop a way to find the compressed representation of data, when data points do not necessarily fall on a line. So, we have to find a line that has the least reconstruction error.

To do this, we need to find the line that has the least reconstruction error.



Let the set of data points be $(x_1, x_2, x_3...., x_n)$, where $x_i \in \mathbb{R}^d$. All of these points do not lie on a particular line, but we have to choose a line in such a way that all the points are close to the line, i.e, the average reconstruction error is minimum. There are n data points. so,

$$\textbf{Average Error(line, dataset)} = \frac{\Sigma_{i=1}^{n}\ error(line, x_i)}{n}$$
$$= \frac{\Sigma_{i=1}^{n}\ length^2(x_i - (x_i^T w)w)}{n}$$

2.2 Maximum Likelihood Estimation

To undestand the theme of Maximum Likelihood Estimation, first let us understand the theme of estimation. Suppose there is a probabilistic mechanism that generates some data about which do not know *something*. Our goal is to find or estimate what we do not know given the data. For example, let us assume that there is a biased coin inside a box, and when we press the box it flips the coin inside it and displays 1 for head and 0 for tail. Now, we do not have any idea about the bias in the coin. Say on flipping the coin twelve time we get the data set 1,0,0,1,1,0,1,1,1,1,1,1. From this data set we have to guess the bias of the coin. We can see that there are 9 heads and 3 tails in this data set. Let us assume that the probability that we would get a head on tossing is **P**. Naturally, the probability of getting a tail on tossing will be **1-P**. Now, we can say for sure that P can never be 1 , since there are three tails in the data set. If **P** was 1 then no tail would have occurred. Similarly, P can not be 0 either. However, **P** can be any value in the range (0,1). By looking at the data set, we can take one guess that **P**= $\frac{3}{4}$ since head occurs 9 out of 12 times in the data set. But we can not be sure of that since the **P** we have guessed is solely based on the data set. Maybe if we had taken more data, we would have got a different guess.

Estimation is also a type of unsupervised learning. What we are basically doing in estimation is:

- Observing the data.
- Assuming a probabilistic model that generates the data.
- Estimating unknown parameters using this data.

We make two assumptions here. The observations are:

- Independent, i.e. the probability of occurrence of a certain data point does not depend on the other data points.
- Identically distributed, i.e one probability distribution should adequately model all values we observe in a sample.

Now that we got an idea about how estimation works , let us move on to maximum likelihood estimation. Let us suppose that we have a random sample $X_1, X_2, ..., X_n$ whose assumed probability distribution depends on some unknown parameter θ. Our primary goal here will be to find a point estimator $u(X_1, X_2, ..., X_n)$, such that $u(x_1, x_2, ...x_n)$ is a "good" point estimate of θ, where $x_1, x_2, ...x_n$ are the observed values of the random sample.

It seems reasonable that a good estimate of the unknown parameter would be the value of θ that maximizes the probability, that is, the likelihood of getting the data we observed. In statistics the terms *probability* and *likelihood* have slightly different meanings.

- *Probability* refers to the chance that a particular outcome occurs based on the values of parameters in a model.
- *Likelihood* refers to how well a sample provides support for particular values of a parameter in a model.

Suppose we have a coin that is assumed to be fair. If we flip the coin one time, the probability that it will land on heads is 0.5.

Now suppose we flip the coin 100 times and it only lands on heads 17 times. We would say that the likelihood that the coin is fair is quite low. If the coin was actually fair, we would expect it to land on heads much more often.

Let us see how to implement the Maximum Likelihood Estimation method in practice with the coin-toss example that we saw earlier. Let us take a data set 1,0,1,1 for four simultaneous tosses. As we already defined, **P** is the probability of getting heads. Also, all these four observations are independent events. So, the probability of this data set occurring will be: P*(1-P)*P*P $= P^3(1-P)$.

Fischer's Principle of Maximum Likelihood

$$\mathbf{L}(\hat{p};(x_1,x_2,x_3..x_n))=\mathbf{P}((x_1,x_2,x_3..x_n);p)$$

$\hat{p}$ is the estimated parameter and p is the underlying parameter or the actual parameter for which the data set is produced.

Since $x_1, x_2...., x_n$ are independent: $\mathbf{P}((x_1,x_2,x_3..x_n);p) = \mathbf{P}(x_1;p)$*$\mathbf{P}(x_2,p)$*.......$\mathbf{P}(x_n;p)$

If the x_i are independent Bernoulli random variables with unknown parameter p, then the probability mass function of each x_i is:

$$f(x_i;p) = p^{x_i}(1-p)^{1-x_i}$$

For $x_i = 0$ or 1 and $0 < p < 1$, the, Likelihood Function L(p) by definition is:

$$\mathbf{L}(\hat{p};(x_1,x_2,x_3..x_n))= \prod_{i=1}^{n} p^{x_i}(1-p)^{(1-x_i)}$$

When x_i= 1, $p^{x_i}(1-p)^{(1-x_i)}$ =$p^1(1-p)^{(1-1)}$ = p.
When x_i= 0, $p^{x_i}(1-p)^{(1-x_i)}$ =$p^0(1-p)^{(1-0)}$ = 1-p.

$$\text{So, } \prod_{i=1}^{n} p^{x_i}(1-p)^{(1-x_i)}$$
$$= (p^{x_1}(1-p)^{(1-x_1)}) * (p^{x_2}(1-p)^{(1-x_2)})...... * (p^{x_n}(1-p)^{(1-x_n)})$$
$$= p^{(x_1+x_2....+x_n)} * (1-p)^{((1-x_1)+(1-x_2)+...(1-x_n))}$$
$$= p^{\sum_{i=1}^{n} x_i}.(1-p)^{n-\sum_{i=1}^{n} x_i}$$

Now, our estimator would be that argument of p which maximizes our Likelihood function or maximizes the probability of getting the given data set.

$$\hat{p}_{ML} = \underset{\mathbf{p}}{\mathbf{arg\ max}} \left(\prod_{i=1}^{n} p^{x_i}(1-p)^{(1-x_i)}\right)$$
$$= \underset{\mathbf{p}}{\mathbf{arg\ max}} \left(\mathrm{p}^{\sum_{i=1}^{n} x_i}.(1-p)^{n-\sum_{i=1}^{n} x_i}\right)$$

Since logarithm function is a monotonically increasing function, the value of p that maximizes the natural logarithm of the likelihood function also maximizes the Likelihood function **L**(p).

$$\underset{\mathbf{p}}{\mathbf{arg\ max}}\ \log\left((\mathrm{p}^{\sum_{i=1}^{n} x_i}.(1-p)^{(n-\sum_{i=1}^{n} x_i)}\right)$$
$$= \underset{\mathbf{p}}{\mathbf{arg\ max}}\ \log(p^{\sum_{i=1}^{n} x_i}) + \log((1-p)^{(n-\sum_{i=1}^{n} x_i)})$$
$$= \underset{\mathbf{p}}{\mathbf{arg\ max}}\ \left(\sum_{i=1}^{n} x_i\right)\log(p) + \left(n-\sum_{i=1}^{n} x_i\right)\log(1-p)$$

We know for getting the maximum value we have to take the first derivative of the function and equate it with 0.

Taking the derivative of the log likelihood function and setting it to 0,

$$\frac{d((\sum_{i=1}^{n} x_i)\log(p)+(n-\sum_{i=1}^{n} x_i)\log(1-p))}{dp} = 0$$
$$\implies \frac{\sum_{i=1}^{n} x_i}{p} - \frac{n-\sum_{i=1}^{n} x_i}{1-p} = 0$$
$$\implies (1-p)\left(\sum_{i=1}^{n} x_i\right) - p\left(n-\sum_{i=1}^{n} x_i\right) = 0$$
$$\implies \sum_{i=1}^{n} x_i - p\sum_{i=1}^{n} x_i - pn + p\sum_{i=1}^{n} x_i = 0$$
$$\implies \sum_{i=1}^{n} x_i - pn = 0$$
$$\implies \hat{p} = \frac{\sum_{i=1}^{n} x_i}{n}$$

$\hat{p}$ tells us that it is an estimate of p. Here, if we calculate for the data set 1,0,1,1, we will get the value of $\hat{p}$ as $\frac{3}{4}$, which is a good estimate of p.

2.3 Softmax function

The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

The Softmax Formula is as follows:

$$\sigma(\vec{Z})_i = \frac{e^{z_i}}{\sum_{i=1}^{K} e^{z_i}}$$

All the z_i values are the elements of the input vector and can take any real value. The term on the bottom of the formula is the normalization term which ensures that all the output values of the function will sum to 1, thus constituting a valid probability distribution.

An example for better understanding:

Let us assume we have an array of three real values. These values could typically be the output of a machine learning model such as a neural network. We want to convert the values into a probability distribution.

$$\begin{bmatrix} 8 \\ 5 \\ 0 \end{bmatrix}$$

Let us calculate the $e^{z_i}s$

$$e^{z_1} = e^8 = 2981$$
$$e^{z_2} = e^5 = 148.4$$
$$e^{z_3} = e^0 = 1$$

These values do not look like probabilities yet. Note that in the input elements, although 8 is only a little larger than 5, 2981 is much larger than 148 due to the effect of the exponential. We can obtain the normalization term, the bottom half of the Softmax equation, by summing all three exponential terms:

$$\sum_{i=1}^{K} e^{z_i} = e^{z_1} + e^{z_2} + e^{z_3} = 2981 + 148.4 + 1 = 3130.4$$

Finally, dividing by the normalization term, we obtain the softmax output for each of the three elements. Note that there is not a single output value because the softmax transforms an array to an array of the same length, in this case 3.

$$\sigma(\vec{Z})_1 = \frac{2981}{3130.4} = 0.9523$$
$$\sigma(\vec{Z})_2 = \frac{148.4}{3130.4} = 0.0474$$
$$\sigma(\vec{Z})_3 = \frac{1}{3130.4} = 0.0003$$

It is informative to check that we have three output values which are all valid probabilities, that is they lie between 0 and 1, and they sum to 1.

We should also note that due to the exponential operation, the first element, the 8, has dominated the softmax function and has squeezed out the 5 and 0 into very low probability values.

2.4 A Brief Idea of Gradient Descent

Gradient Descent is one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates.

The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.

Let us understand how Gradient Descent actually works. The gradient descent algorithm behaves similar to the linear regression algorithm but instead of a straight line, it is based on a convex function. On the convex function, **f(x)**, we will consider an arbitrary point, whose slope we will find with the help of the first derivative of **f(x)** at that arbitrary point The slope will inform the updates to the parameters—i.e. the weights and bias. The slope at the starting point will be steeper, but as new parameters are generated, the steepness should gradually reduce until it reaches the lowest point on the curve, known as the point of convergence. We will elaborate more on the terms *weights* and *biases* in the subsection of Neural Networks.

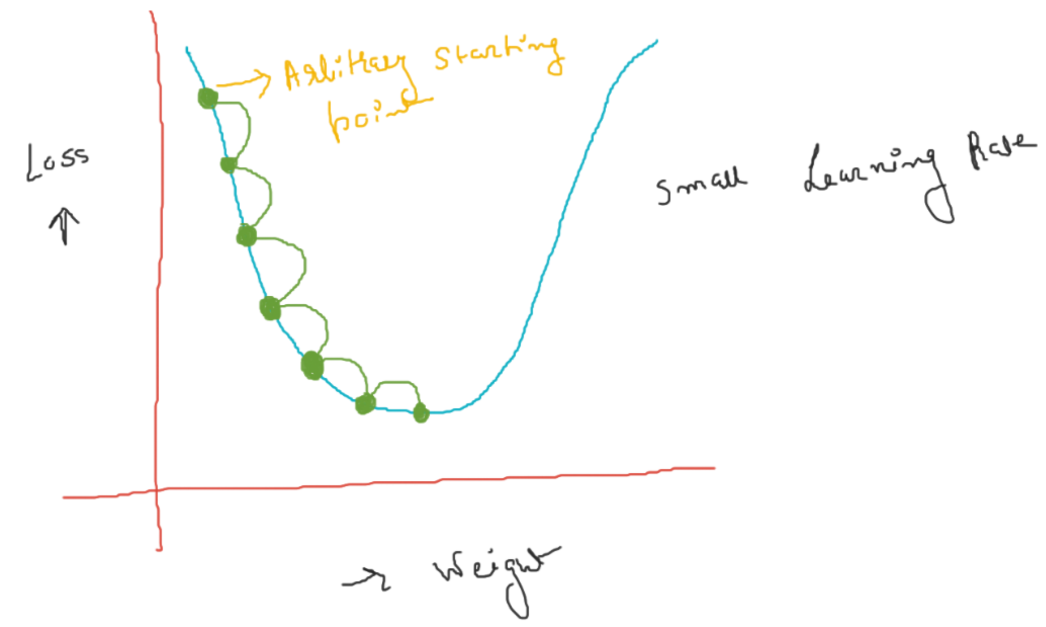
Similar to finding the line of best fit in linear regression, the goal of gradient descent is to minimize the cost function, or the error between predicted and actual y. In order to do this, it requires two data points—a direction and a learning rate. These factors determine the partial derivative calculations of future iterations, allowing it to gradually arrive at the local or global minimum (i.e. point of convergence).

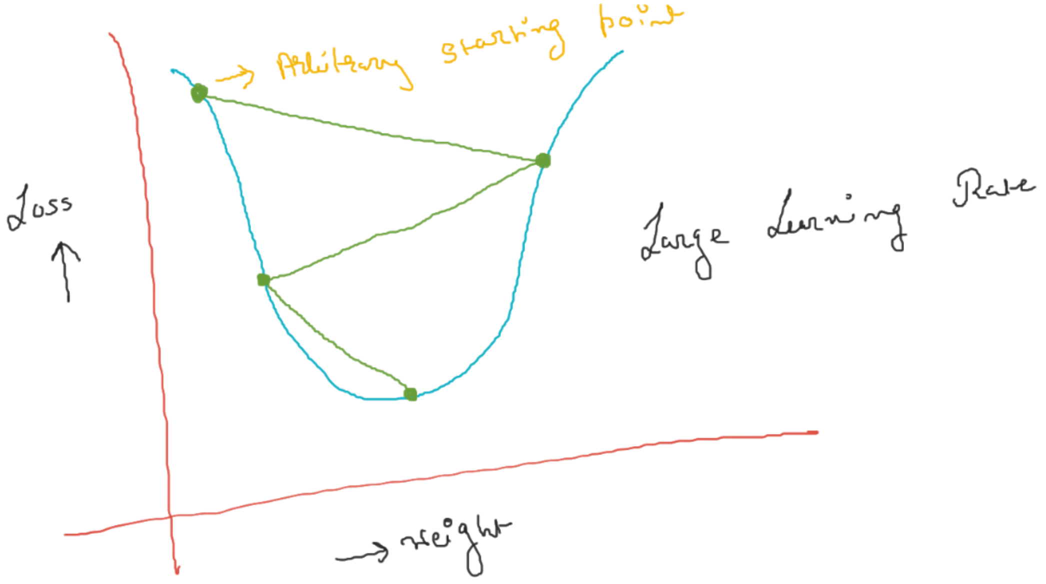
- **Learning Rate**: (also referred to as step size or the alpha) is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function. High learning rates result in larger steps but risks overshooting the minimum. Conversely, a low learning rate has small step sizes. While it has the advantage of more precision, the number of iterations compromises overall efficiency as this takes more time and computations to reach the minimum.
- **The cost (or loss) function** measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at zero. At this point, the model will stop learning. Additionally, while the terms, cost

function and loss function, are considered synonymous, there is a slight difference between them. It's worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

Gradient Descent Formula is represented as:

$$a_{n+1} = a_n - \alpha.\nabla F(a_n) \qquad [\alpha \text{ is the learning rate.}]$$

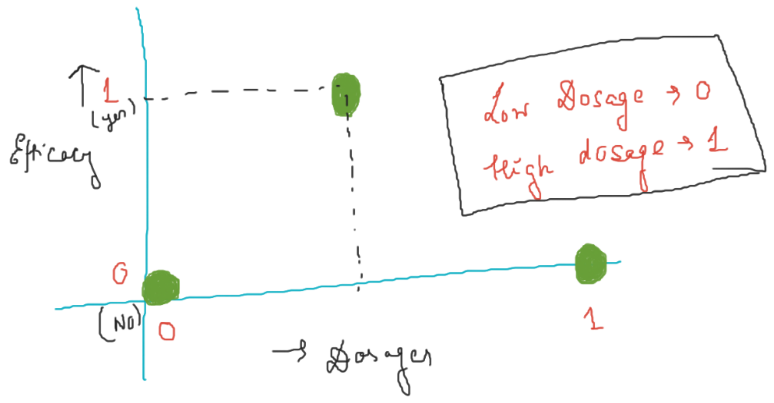




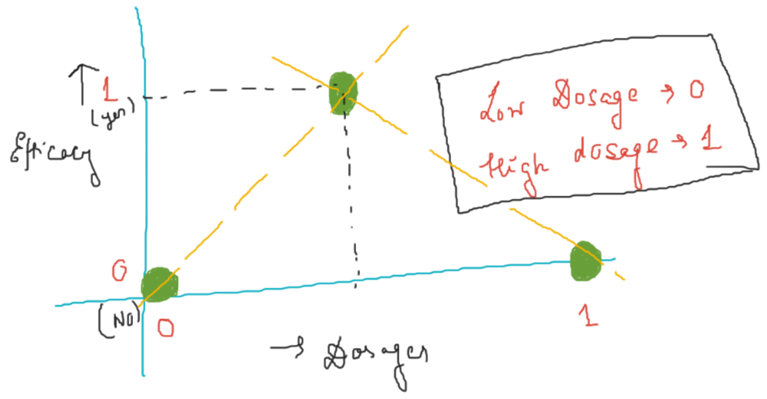
We will see a numerical example of how Gradient Descent is used to optimize parameters in the subsection of Neural Networks. It will make our understanding of Gradient Descent clearer.

2.5 Neural Networks - An Overview

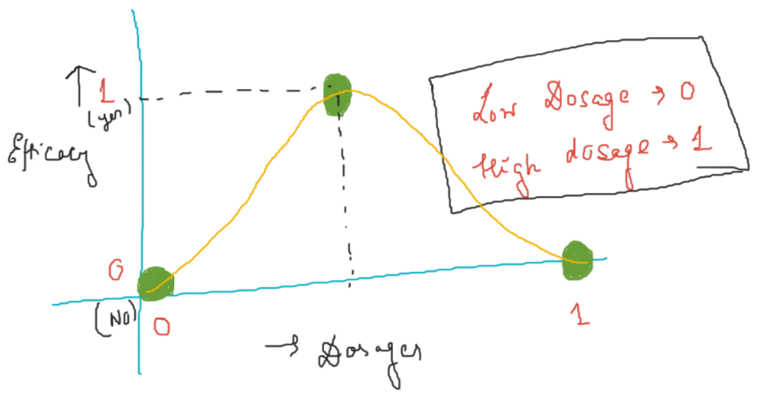
Let us start our understanding of neural networks with the help of an example rather than some formal definitions. Let us suppose that a particular drug was designed to treat a certain illness and it was tested on three different groups with three different dosages. Let us say that after the experiment we deduced that the low and high dosages were ineffective but the medium dosage was effective, which can be seen in the graph below:



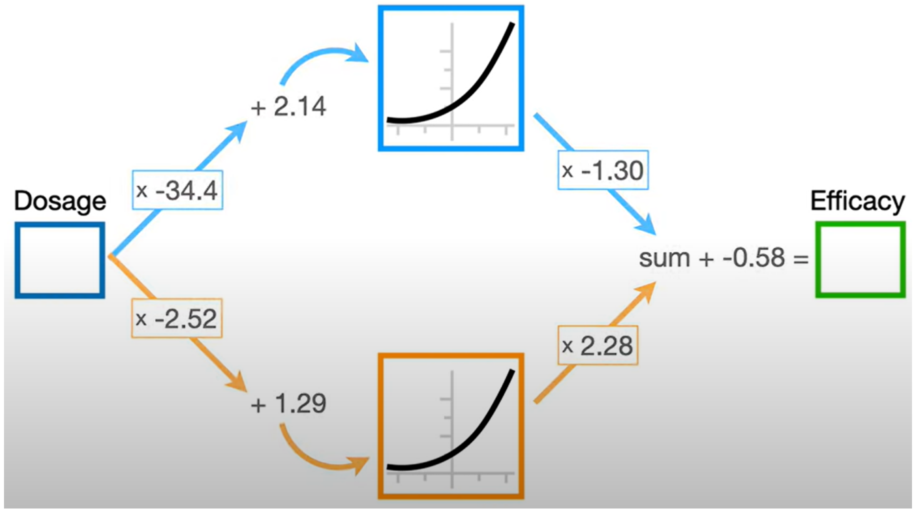
Now that we have this data, we would like to use it to predict whether a future dosage would be effective. Generally for predictions we use a linear regression model where we try to fit a straight line to the data points. However, here we cannot make use of a straight line because no matter how we rotate the line, it can only accurately predict two of three dosages.



However a curve like the one below will fit the data.



Neural Networks help us do exactly that. Let us try to understand how this is done in a simplified way. We are going to consider the simple data set of drug dosages and understand how the neural network illustrated below helps in fitting of the data.



A neural network consists of nodes and connections between the nodes. We should note the numbers along each connection, called the parameters which are estimated when the neural network is fit to the data by a procedure called backpropagation which we will soon discuss. The neural network starts with unknown parameter values which are estimated when we fit the neural network to a data set with the help of backpropagation. For now let us assume that we have already fit this neural network to the dataset,i.e. we have already estimated the parameters. Let us discuss the components of a neural network:

- **Input Layer:**The input layer is first. The data will be accepted by this layer and forwarded to the remainder of the network. This layer allows feature input. It feeds the network with data from the outside world; no calculation is done here; instead, nodes simply transmit the information (features) to the hidden units.
- **Hidden Layer:**Since they are a component of the abstraction that any neural network

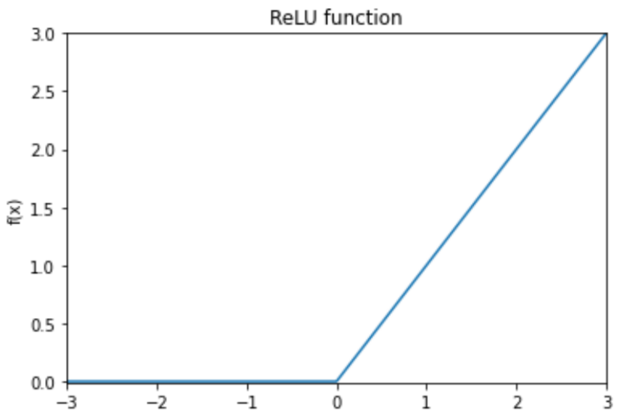
provides, the nodes in this layer are not visible to the outside world. Any features entered through to the input layer are processed by the hidden layer in any way, with the results being sent to the output layer. The concealed layer is the name given to the second kind of layer. For a neural network, either there are one or many hidden layers. The number inside the example above is 1.

- **Output Layer:**This layer raises the knowledge that the network has acquired to the outside world. The output layer is the final kind of layer The output layer contains the answer to the issue. We receive output from the output layer after passing raw photos to the input layer.

In the above example, Dosage is the input layer, efficacy is the output layer and the rest is the hidden layer. In the hidden layer, we can observe two nodes with bent curves inside them. These curves are called activation functions. They are the building blocks of the curve that finally fits the data. We will see how that is done but first let us learn about the types of activation functions that are used.

- **ReLU function:**There are a number of widely used activation functions in deep learning today. One of the simplest is the rectified linear unit, or ReLU function, which is a piecewise linear function that outputs zero if its input is negative, and directly outputs the input otherwise:

$$f(x)=\max(0,x)$$



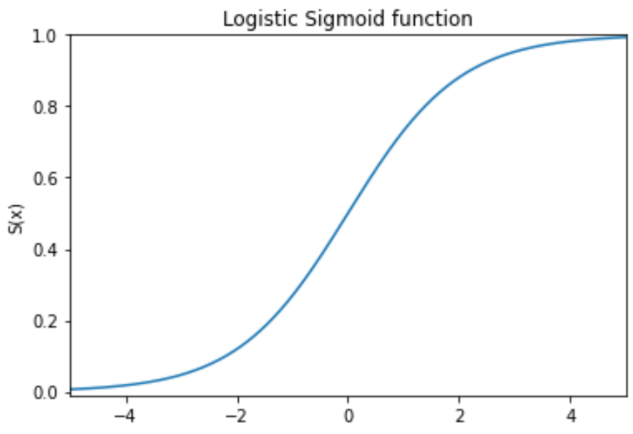
$\frac{d(f(x))}{dx}$ is 1 when $x > 0$, 0 when $x < 0$ and undefined at 0.

In practice the derivative at x = 0 can be set to either 0 or 1 or it will cause problem during backpropagation.

- **Logistic Sigmoid Function:**

$$S(x) = \frac{e^x}{e^x+1}$$

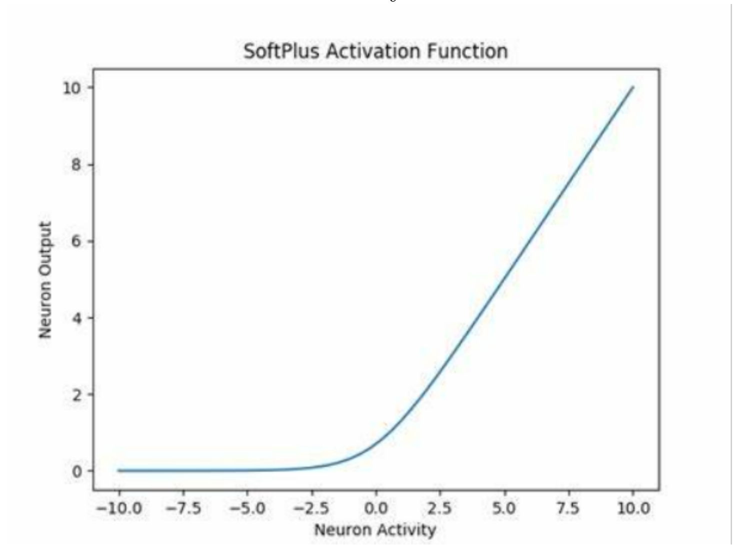
The logistic sigmoid function has the useful property that its gradient is defined everywhere, and that its output is conveniently between 0 and 1 for all x. The logistic sigmoid function is easier to work with mathematically, but the exponential functions make it computationally intensive to compute in practice and so simpler functions such as ReLU are often preferred.



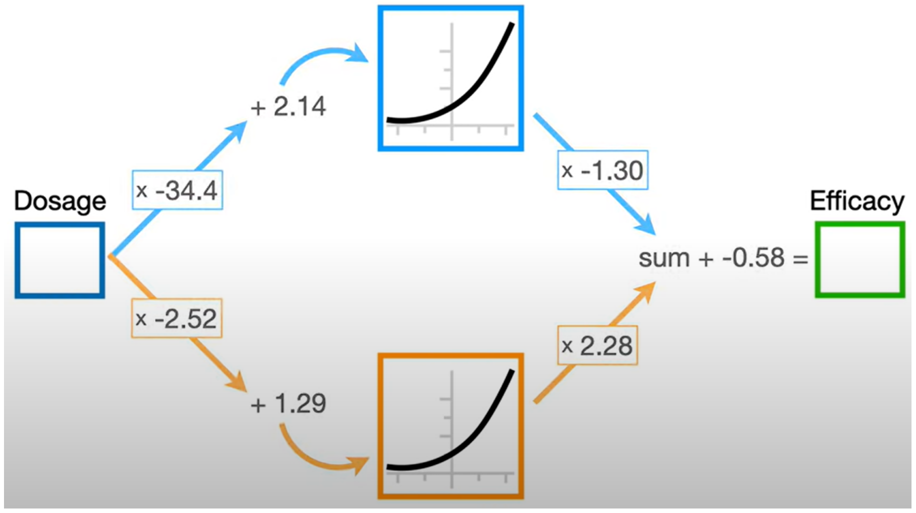
- **Softplus Function:**

$$f(x) = log(1 + e^x)$$

The output of a softplus function is always positive. This activation function is a smooth continuous version of reluLayer.



Now, let us see how to work with neural networks. we will consider the neural network that we saw for fitting the dosage data set.



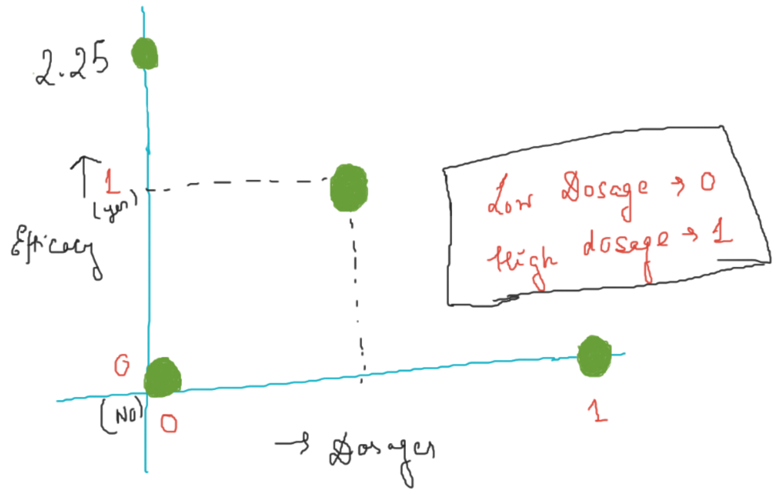
Let us input a value 0 which represents the lowest dosage in the input layer of the neural network. We can see there are two nodes in the hidden layer, let us first compute till the upper node, that is, the path of the blue line. First we will multiply 0 with -34.4 and then add 2.14 to the result according to the neural network. The parameters that we multiply are called weights and the parameters we add are called biases. After doing the computations above, we get the x-coordinate .

$$x = 0 * (-34.4) + 2.14 = 2.14$$

We will get the y-coordinate by putting the x-coordinate value in the activation function equation. Here, we are using the softplus function as the activation function. Thus,

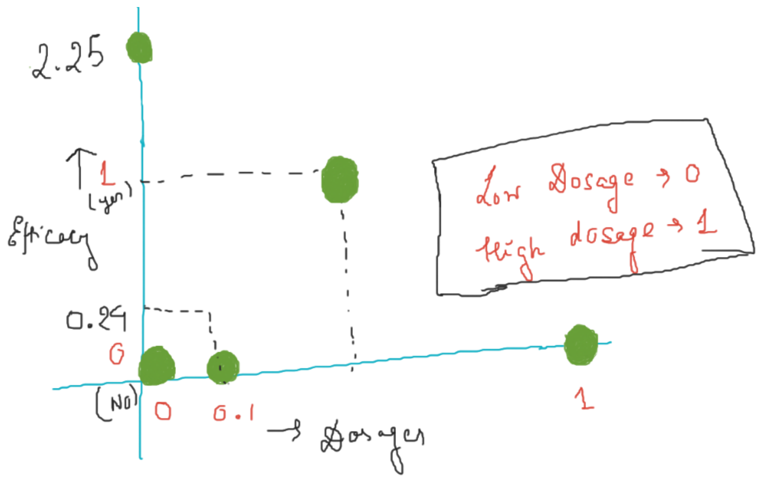
$$\begin{gathered} f(x) = log(1 + e^{x}) \\ = f(2.14) = log(1 + e^{2.14}) \\ = 2.25 \end{gathered}$$

So, for x=0, let us plot y=2.25 on the dosage graph:

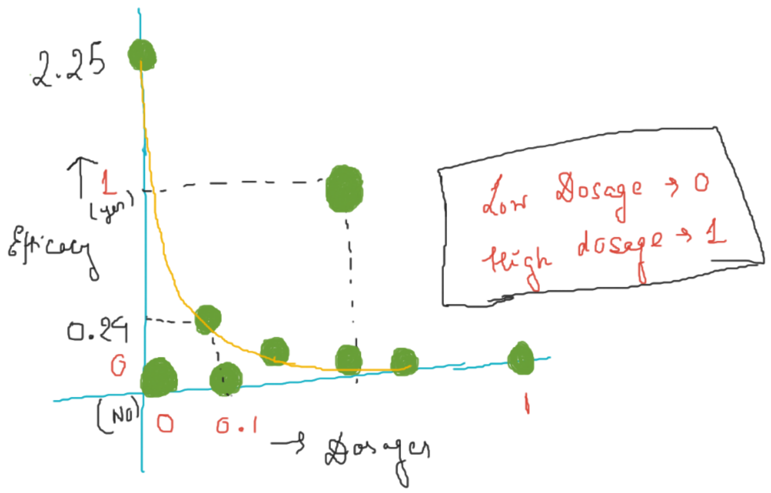


Now let us take the input dosage as 0.1. On performing similar calculations:

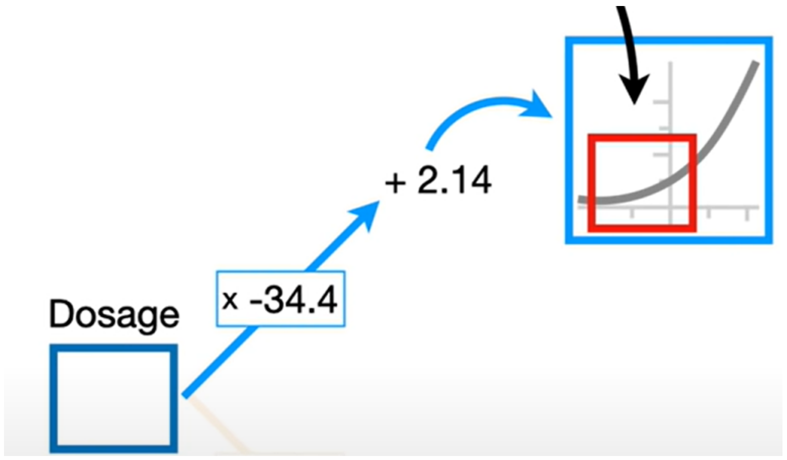
$$0.1 * (-34.4) + 2.14 = -1.3$$
$$f(-1.3) = log(1 + e^{-1.3})$$
$$= 0.24$$



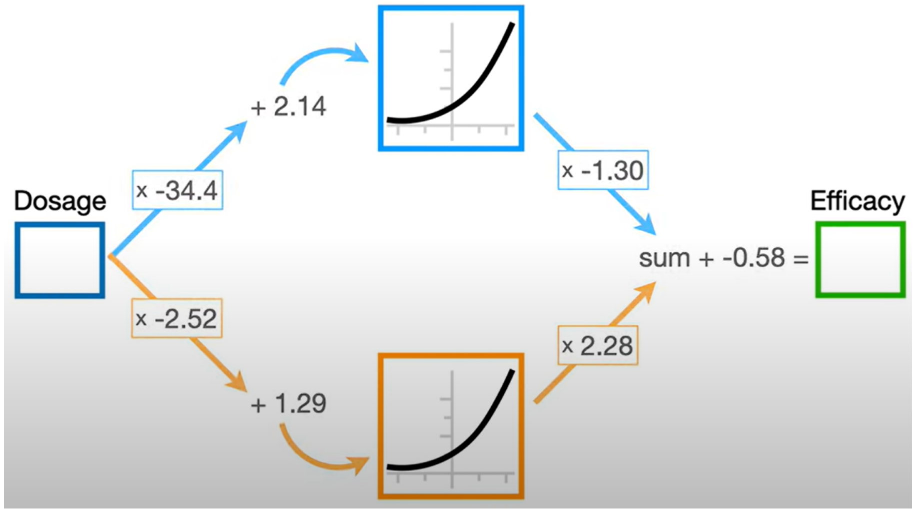
If we continue increasing the dosage value from 0 to 1 in a similar way, we will end up with the following yellow curve. (We may check by plugging more points for better accuracy of drawing the curve.)



NOTE: The full range of dosage values from 0 to 1 , corresponds to a relatively narrow range of values from the activation function.

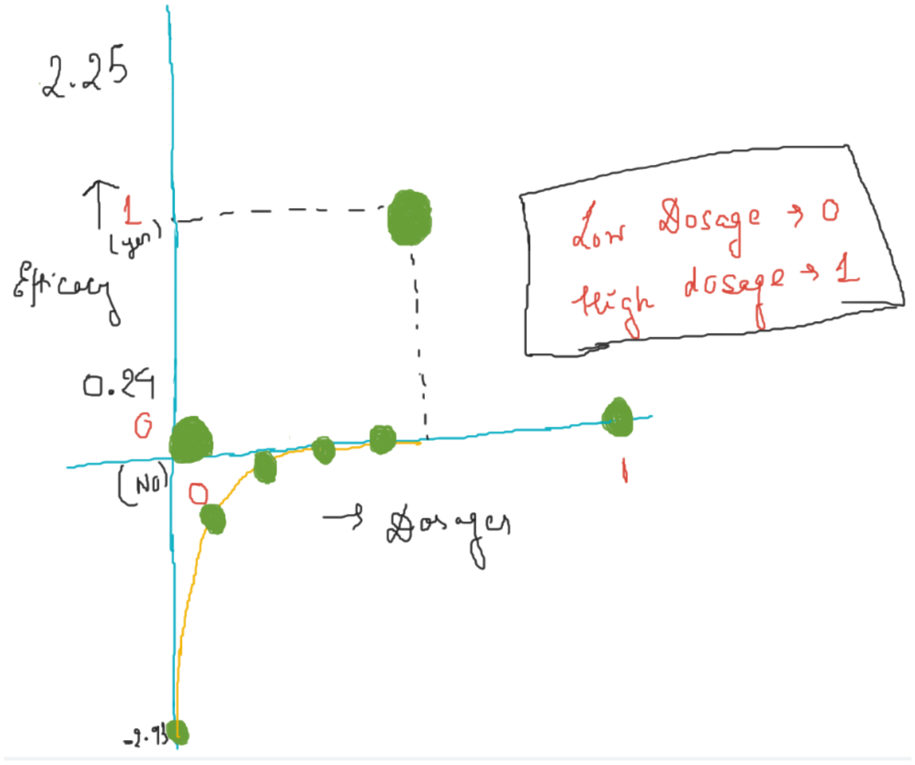


Only the corresponding y-axis values in the red box are used to make the yellow curve that we saw earlier.

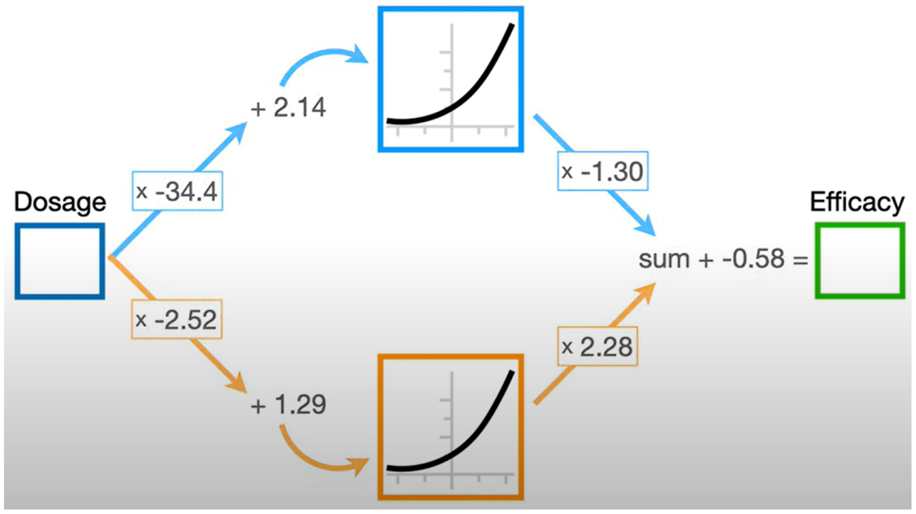


Now we will scale the y-axis values by -1.30, i.e., we will multiply they-coordinates with -1.30. For example for dosage = 0, we got the y-coordinate as 2.25 and $2.25 * (-1.30) = -2.93$.

Likewise, we multiply all other y-axis values to the yellow curve to get the following yellow curve:



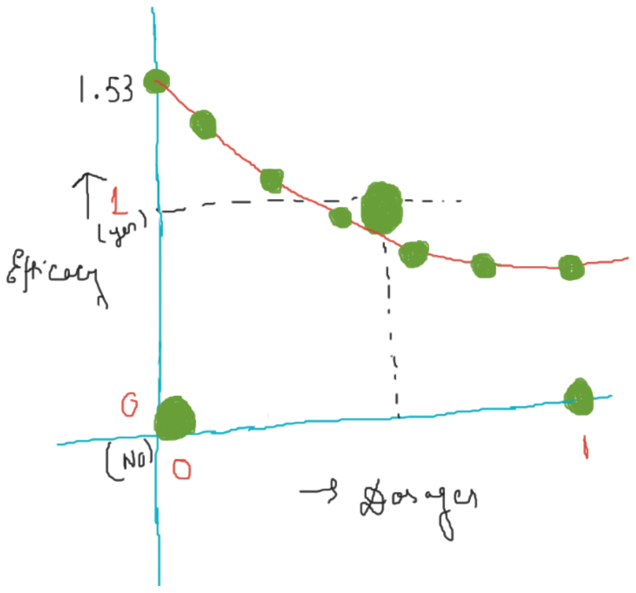
Now, we will focus on the connection from the input node to the bottom node in the hidden layer.



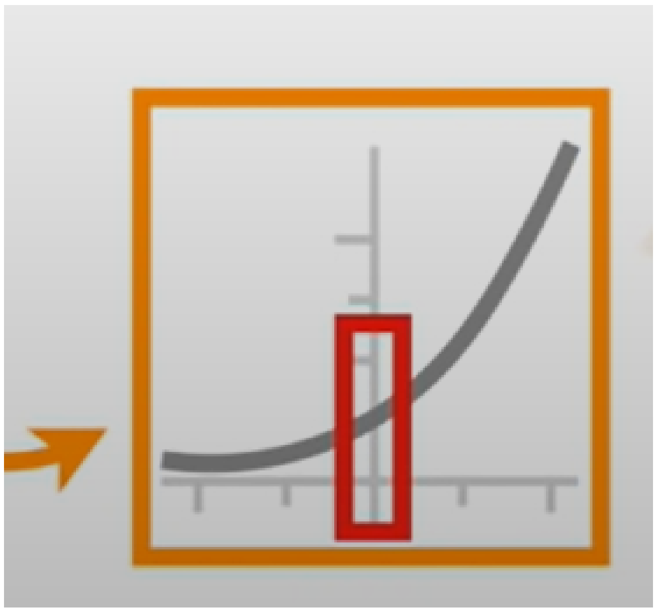
Let us consider an input dosage of 0.

$$0 * (-2.52) + 1.29 = 1.29$$
Again we have the softplus function so,
$$f(1.29) = log(1 + e^{1.29})$$
$$= 1.53$$

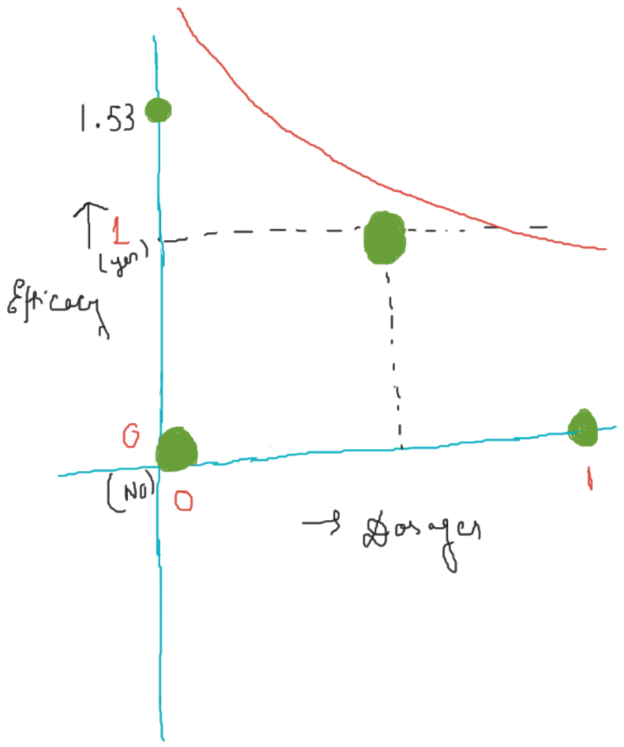
So for x=0, we will get y=1.53 for the bottom node in the hidden layer. Likewise, for values between 0 and 1, we will get a red curve like this:



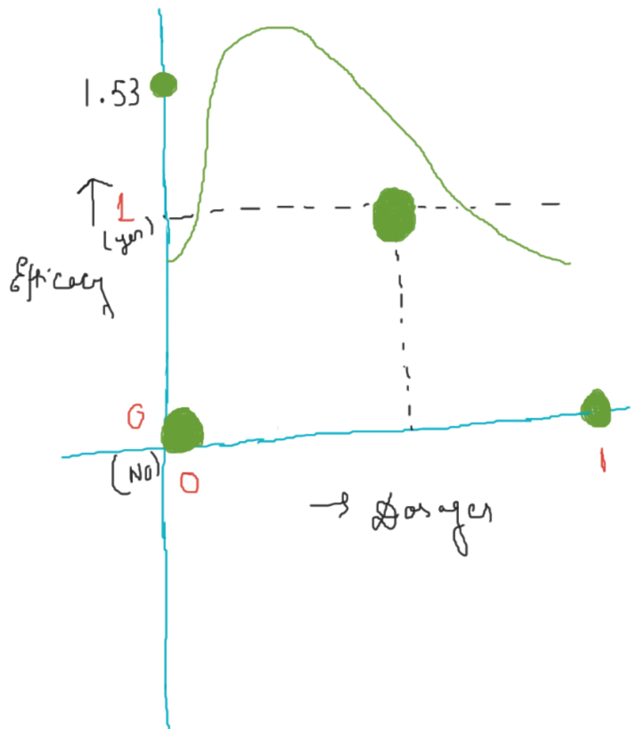
NOTE: The full range of dosage values from 0 to 1 , corresponds to a relatively narrow range of values from the activation function.



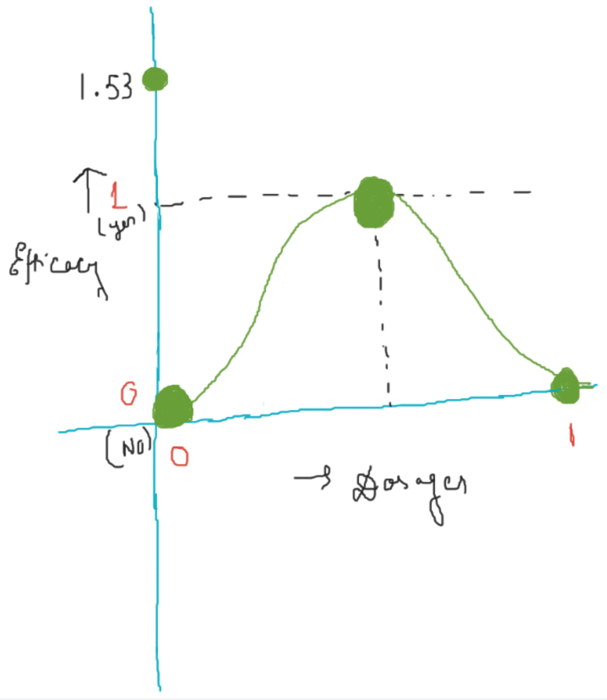
After scaling the y-coordinates by multiplying them with 2.28, we will get the following red curve for the bottom node.



From the neural network we can see that we are supposed to find the sum of the y coordinates given by the upper and the lower nodes. When we do so, we get the following green curve:



After subtracting 0.58 from the y-coordinates of the green curve we get the new green curve that fits the data.

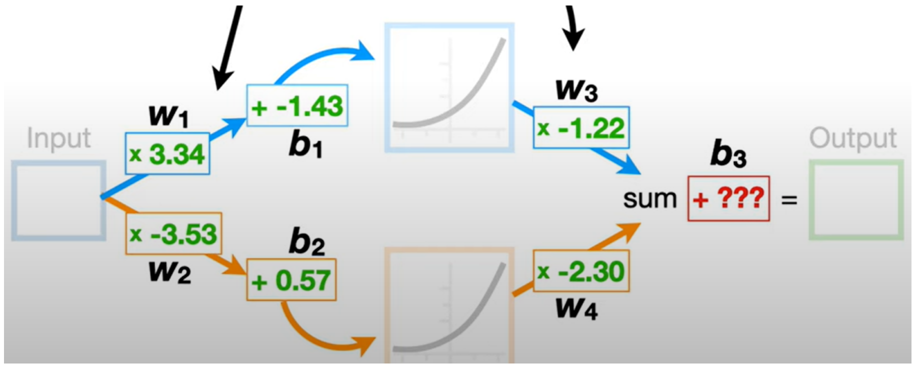


But the most important question that arises is: *Where from are we getting the values for the weights and the biases that are causing the curve to fit the data set?*

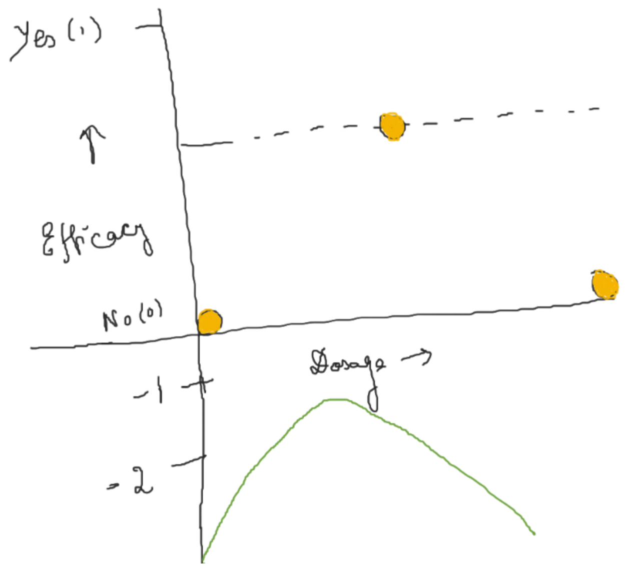
We find the optimal weights and biases by a method known as the Backpropagation.

Backpropagation

Neural Networks start with identical activation functions but using different weights and biases on the connections, it flips and stretches the activation functions into different shapes. So, estimating weights and biases are of utmost importance. We can predict all the weights and biases through backpropagation but let us do it for one parameter for easier understanding. Let us assume that all the parameters are properly optimized except b3. We will optimize b3 by backpropagation:



If we perform all the steps of the neural network, except adding the last bias b3 (steps are discussed above) , we end up with the following curve:



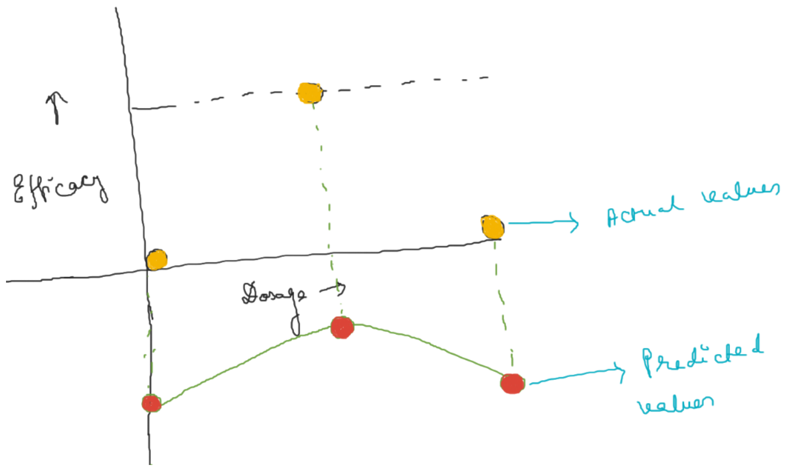
We can see that the curve does not fit the data but if we move the curve up or add some positive bias to the y-coordinates of the curve , we can move it upwards to fit the data. So, our main objective will be to find the optimal b3 to fit the curve. Let us see the algorithm for doing the same.

Algorithm for optimizing b3

STEP-1: We will initialize b3 with any random value, here let us initialize b3 with 0.

STEP-2: We want to find the Sum of Squared Residuals:

$$\text{Residual Value} = \text{Observed Value - Predicted Value}$$

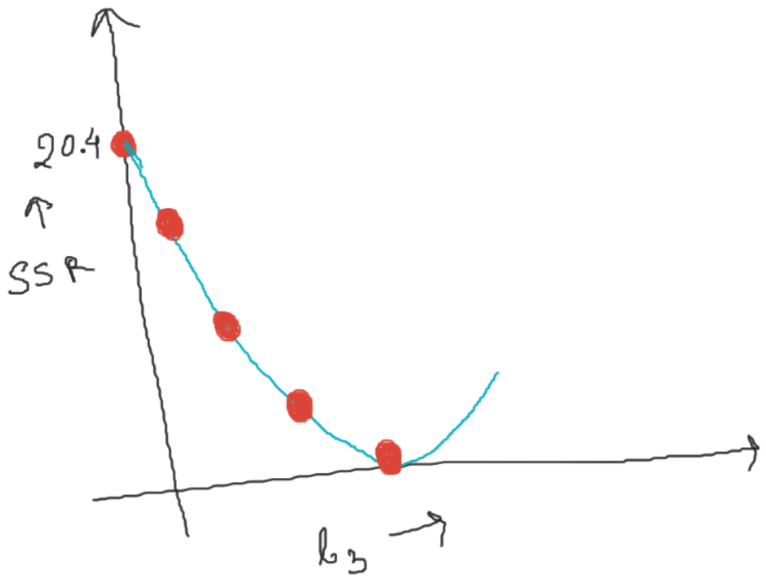


If we plug the values 0, 0.5 and 1 in the input of the neural network respectively when b3 = 0, we will get -2.6, -1.61 and -2.61 respectively as the predicted values. Our observed values on the other hand are 0, 1 and 0 respectively.

$$\text{Sum of Squared Residuals(SSR)} = \sum_{i=1}^{n}(observed_i - Predicted_i)^2$$

Here, n = 3. So, SSR = $(0-(-2.6))^2 + (1-(-1.61))^2 + (0-(-2.61))^2 = 20.4$

STEP-3: Plot SSR v/s b3 graph. When b3 = 0, SSR = 20.4. Our goal is to minimize SSR. If we increase b3. SSR wwill decrease as we can see from the graph. That is where we use gradient descent to optimize the value for b3.



We know that the gradient descent formula is : $a_{n+1} = a_n - \alpha.\nabla F(a_n)$ [α is the learning rate.] Let us consider $\alpha = 0.1$

We need to find $\frac{d(SSR)}{d(b3)}$. We know that,

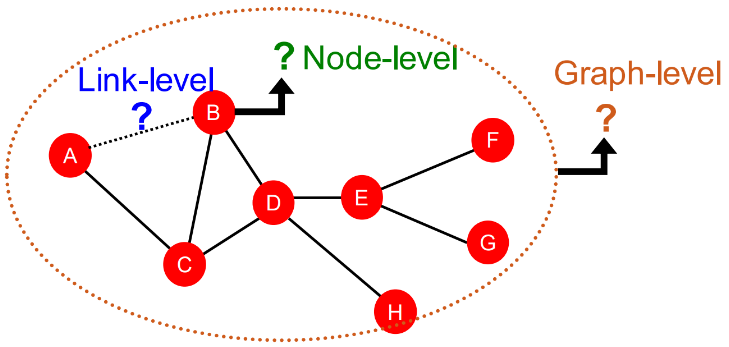
$$\text{Sum of Squared Residuals(SSR)} = \textstyle\sum_{i=1}^{n}(observed_i - Predicted_i)^2$$

Also, curve obtained from top node of hidden layer + curve obtained from bottom node of hidden layer + b3 = The predicted curve that fits the data.
So, $\frac{d(SSR)}{d(b3)} = \frac{d(SSR)}{d(b3)}$

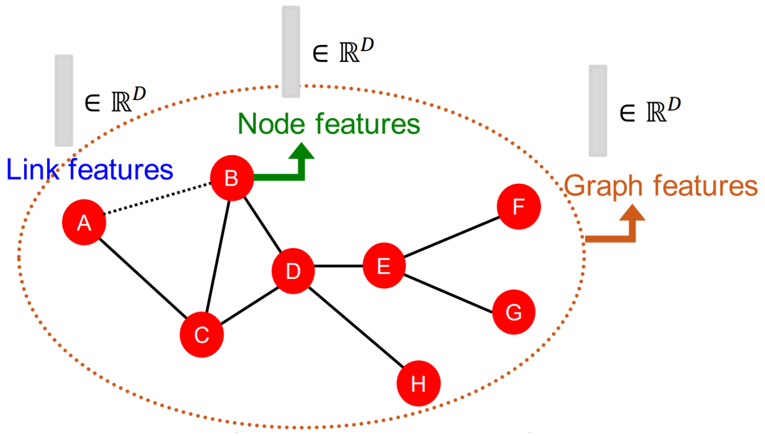
3 Machine Learning on Graphs - Feature Engineering

Machine Learning Tasks on Graphs

- Node-level predictions
- Link-level predictions
- Graph-level predictions



In a traditional Machine Learning Pipeline, we have two steps. In the first step we are going to take our data points, i.e. nodes , links and graphs and represent them with vectors of features and train the classical machine learning models. Then we can apply these models where a new node, link or graph appears and obtain it's features to make predictions.



Train a ML model:

- Logistic Regression
- Random forest
- Neural network, etc.

Apply the model:

- Given a new node/link/graph, obtain its features and make a prediction

Using effective features x over graphs is the key to achieving good model performance.
NOTE: We will use undirected graphs for simplicity.

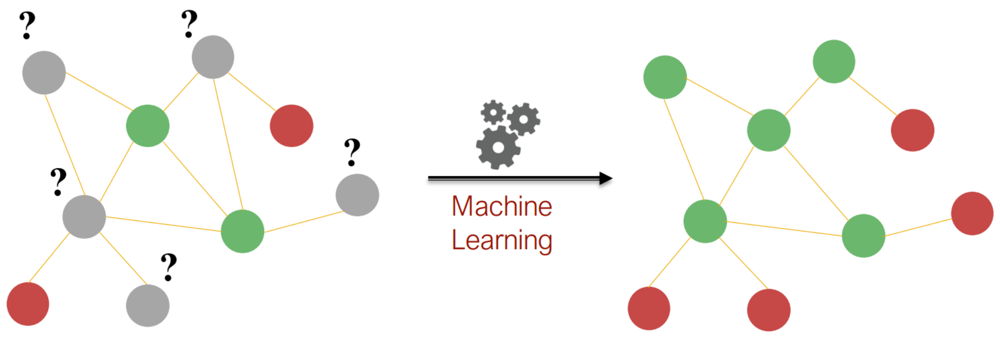
Let us sum up Machine Learning on Graphs as:

GOAL: Make predictions for a set of objects

DESIGN CHOICES:

- **Features**: d-dimensional vectors x.
- **Objects**: Nodes, edges, sets of nodes, entire graphs.
- **Objective function**: What task are we aiming to solve?

We will first discuss about node level predictions then slowly move to edge level predictions and finally we will discuss graph level predictions. Let us understand an example of Node Level Task.



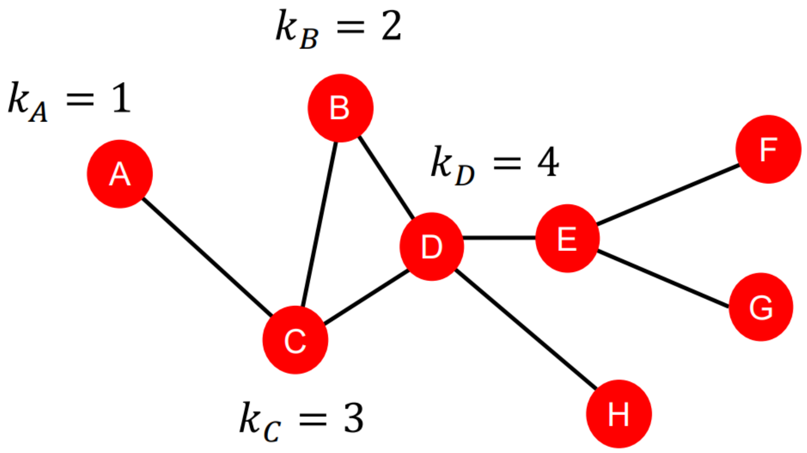
In the above diagram we are given a graph where some nodes are coloured red and green. Given the red and green nodes, our goal is to predict the colour for the uncoloured nodes or the gray nodes. If we observe carefully, we will deduce that the nodes with at least two edges are coloured green and the nodes with one edge is coloured red. So, if we describe the node degree as a structural feature in this graph, then we will be able to learn the model that correctly colours the nodes of the graph.

Now, our goal is to characterize the structure and position of a node in the network. We will discuss four different approaches that will allow us to do this. They are:

- Node Degree
- Node Centrality
- Clustering Coefficient
- Graphlets

3.1 Node Features - Node Degree

The degree k_v of a node v is the number of edges (neighboring nodes) v has.



The drawback of using node degree as a feature is that it treats all the nodes equally. As a result, nodes with same degree may be indistinguishable even if they are in different parts of the network. For example, the nodes C and E have the same degree. So, our classifier will not be able to distinguish between them. Similarly, nodes A, H, F, G all have degree 1. As a result, they will have the same feature values.

So, we could simply say that node degree counts the neighboring nodes without capturing their importance. That is where node centrality c_V, which takes the node importance in a graph into account, comes in.

3.2 Node Features - Node Centrality

In graph analytics, Centrality is a very important concept in identifying important nodes in a graph. It is used to measure the importance (or "centrality" as in how "central" a node is in the graph) of various nodes in a graph. Now, each node could be important from an angle depending on how "importance" is defined. Centrality comes in different flavors and each flavor or a metric defines importance of a node from a different perspective and further provides relevant analytical information about the graph and its nodes. The different types of centrality measures are:

a. Degree Centrality

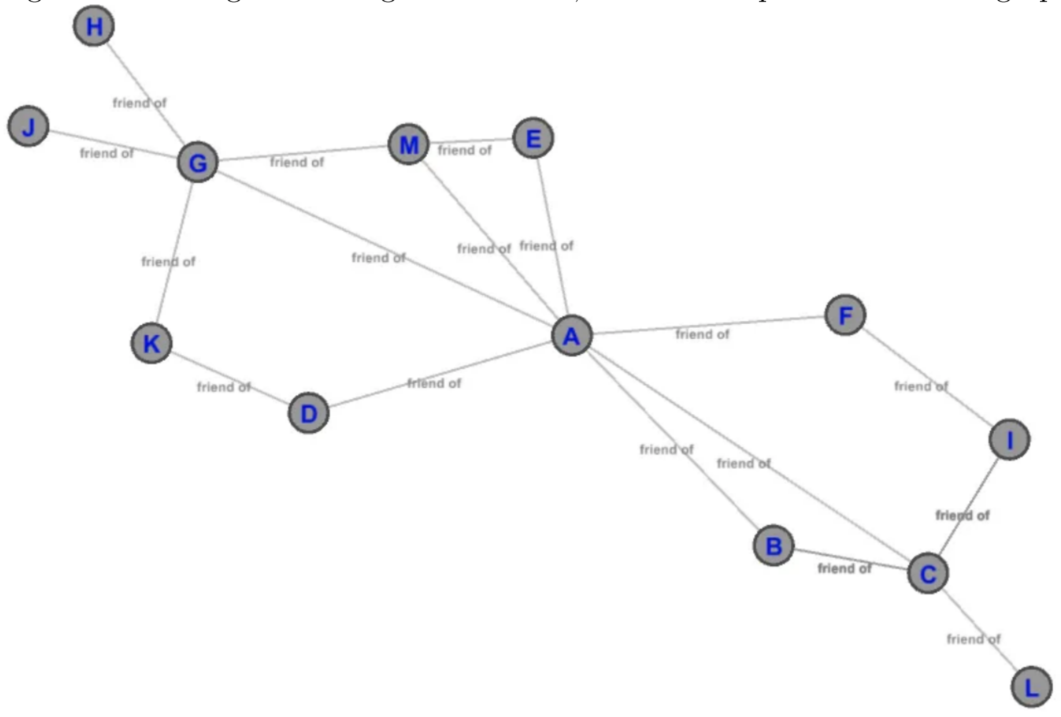
b. Closeness Centrality

c. Betweenness Centrality

d. Eigenvector Centrality

Degree Centrality

Degree Centrality metric defines importance of a node in a graph as being measured based on its degree i.e. the higher the degree of a node, the more important it is in a graph.



Let us look at the sample graph of friends above. The degree centrality of node A is 7, node G is 5, node C is 4 and node L is 1. Mathematically, Degree Centrality is defined as D(i) for a node "i" as below:

$$\mathbf{D(i)} = \Sigma_j \mathbf{m(i,j)}$$

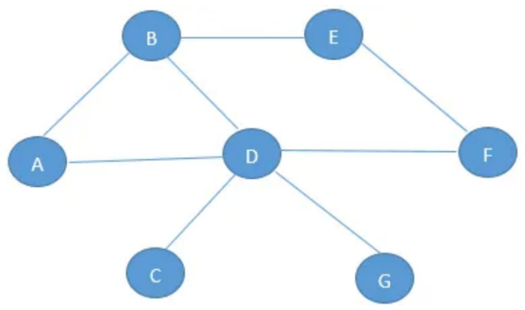
m(i,j) = 1 if there is a link from node i to node j.

Now let us briefly discuss a sample application of degree centrality to the above shown graph of friends. Looking at node A and G, they have a high degree centrality(7 and 5 respectively) and will be ideal candidates if we want to propagate any information to a large part of the network quickly as compared to node L which only has a degree centrality of 1.This information is very useful for creating a marketing or an influencing strategy if a new product or idea/thought has to be introduced in the network. Marketers can focus on nodes such as A,G etc. with high degree centrality to market their product or ideas in the network to ensure higher reach-ability among nodes.

Closeness Centrality

To understand Closeness Centrality, first let us understand the concept of "Geodesic distance" between two nodes in a graph. The Geodesic distance d between two nodes a and b is defined as the number of edges/links between these two nodes on the shortest path(path with minimum number of edges) between them.

Let us look at the graph below:



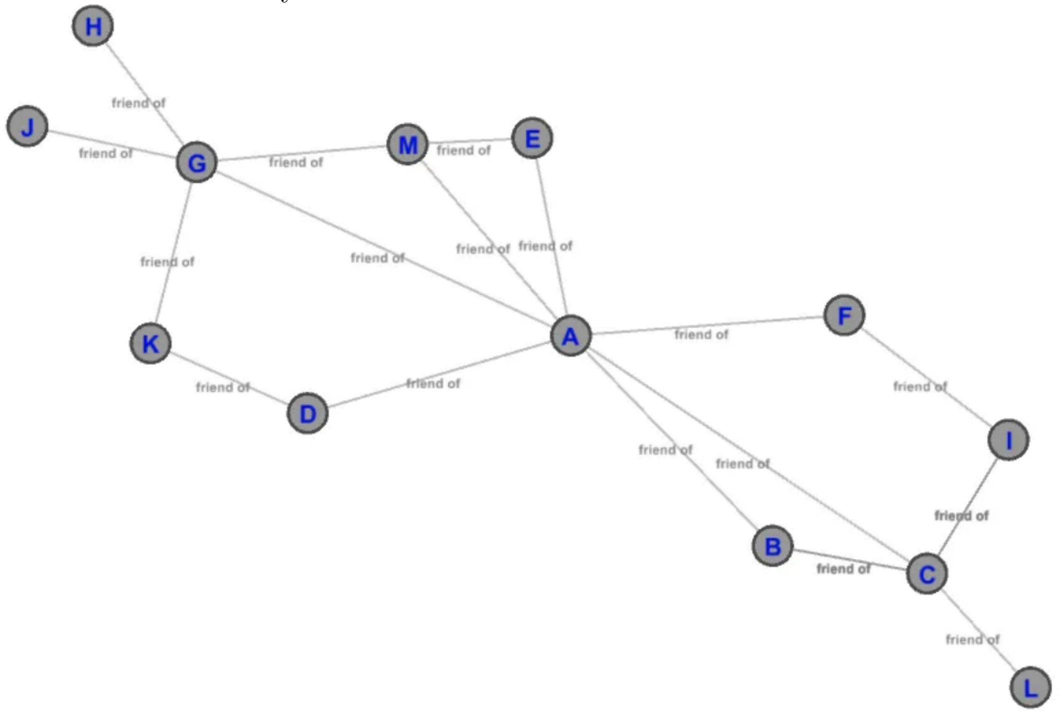
Let us examine the geodesic distance between A and F to further clarify the concept. We can reach F from A by going through B and E or by going through D. However, the shortest path from F to A is through D(2 edges), hence the geodesic distance d(A,F) will be defined as 2 as there are 2 edges between A and F.

Mathematically, Geodesic distance can be defined as below:

d(a , b) = No. of edges between a and b on the shortest path from a to b, if a path exists from a to b.
d(a , b) = 0, if a = b.
d(a , b) = ∞ (Infinity) , if no path exists from a to b.

Further, closeness centrality metric defines the importance of a node in a graph as being measured by how close it is to all other nodes in the graph.For a node, it is defined as the sum of the geodesic distance between that node to all other nodes in the network.

Again, looking at the previously introduced graph of friends in Figure 1 below, we can see that the Closeness centrality of node A is 17 while that of node L is 33.



Mathematically, Closeness Centrality C(i) of a node i in a graph can be defined as below:

$$\mathbf{C(i)}= \frac{1}{\Sigma_j d(i,j)}$$

Let us briefly describe a sample application of Closeness Centrality by examining the friends graph. Now let us suppose that in the friend's graph, each link/edge has a weight (attribute) of 1 minute associated with it, i.e. it would take 1 minute to transmit information from a node to its neighboring node such as A to B or B to C. Now let us suppose we want to send a piece of specific information (information will be different for each node) to each node of the graph and we need to select a node in the graph that can transmit it quickly to all the nodes in the network.

To solve the above problem, we can calculate the Closeness Centrality measure for all the nodes in the network. As we already calculated above for node A, if we select node A, the information can reach all the nodes by traversing 17 edges (i.e starting at A, information can be transmitted to all nodes in 17 minutes in a worst case scenario assuming sequential sends from A) as compared to node L, where it would take 33 minutes to transmit the information to all nodes.Clearly we can see the difference in importance of both the nodes A and L in terms of Closeness Centrality measure.

Betweenness Centrality

The third flavor of centrality we are going to discuss is known as "Betweenness Centrality" (BC). This metric defines and measures the importance of a node in a network based upon how many times it occurs in the shortest path between all pairs of nodes in a graph.

Mathematically, Betweenness Centrality B(i) of a node i in a graph is defined as below:

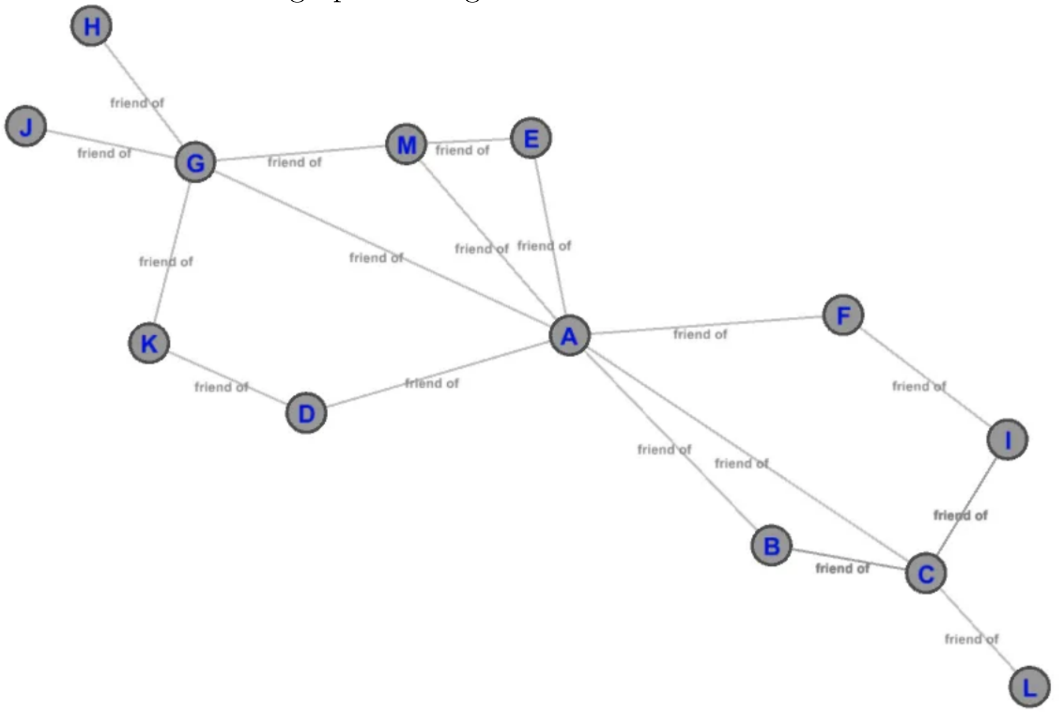
$$\mathbf{B(i)}= \Sigma_{a,b}\frac{g_{aib}}{g_{ab}}$$

a,b is any pair of nodes in the graph.

g_{aib} is the number of shortest paths from node "a" to node "b" passing through "i".

g_{ab} is the number of shortest paths from node "a" to node "b".

Let us consider our friend's graph once again:



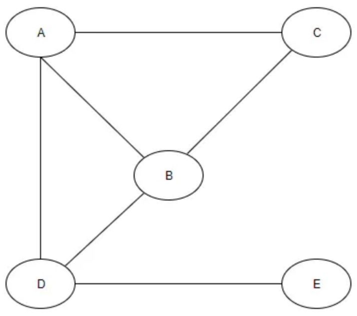
Looking at node A, we can observe that it lies on the shortest path between the following pair of nodes : (D,M), (D,E),(G,C),(G,B),(G,F),(G,I),(K,C),(D,C) etc. and thus has the highest BC among all other nodes in the graph. We can also observe that both nodes G and C also have high Betweenness Centralities (BCs) as compared to other nodes (except A) in the graph

As discussed, if we look at our friends graph above (Figure 6), node A has a very high BC. If we were to remove it, it would lead to huge disruption in the network as there would be no way for nodes J,H,G,M,K,E,D to communicate with nodes F,B,C,I,L and vice versa and we would end up with two isolated sub graphs. This understanding marks the importance of nodes with high BCs.

Eigenvector Centrality

The last flavor of centrality that we will be exploring is known as the Eigen Vector Centrality. This metric measures the importance of a node in a graph as a function of the importance of its neighbors. If a node is connected to highly important nodes, it will have a higher Eigen Vector Centrality score as compared to a node which is connected to lesser important nodes.

Let's look at the graph given below to further explain the concept:



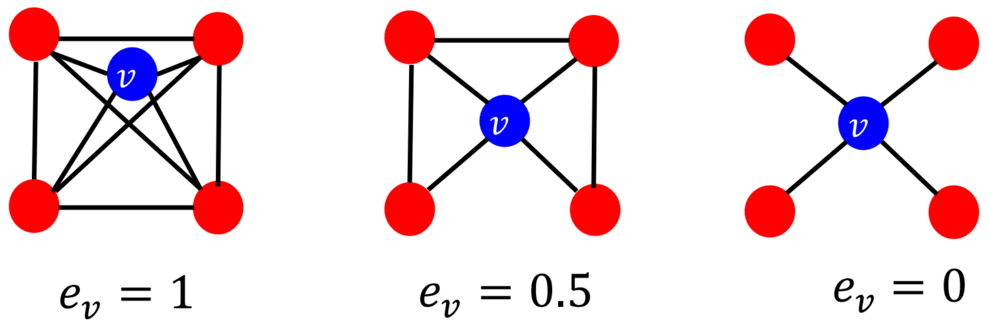
will add more later......

3.3 Node Features - Clustering Coefficient

Clustering Coefficient is basically the measure of how connected a node v's neighboring nodes are. Mathematically, it can be represented as:

$$e_v = \frac{No.\ of\ edges\ among\ neighbouring\ nodes}{\binom{k_v}{2}}$$

k_v is the degree of the node v. So, k_v is basically the number of neighbouring nodes v is connected to. So, $\binom{k_v}{2}$ is the node pairs among the neighbouring nodes. Below are some examples:



3.4 Node Features - Graphlets

4 An overview of the Node2Vec Paper by Jure Lekstovec and Aditya Grover