CS06 : AI-ML
Mid Term Report

Rakshit Rane

21 June 2024

CONTENT -

CHAPTER 1

Supervised Learning : Regression and Classification

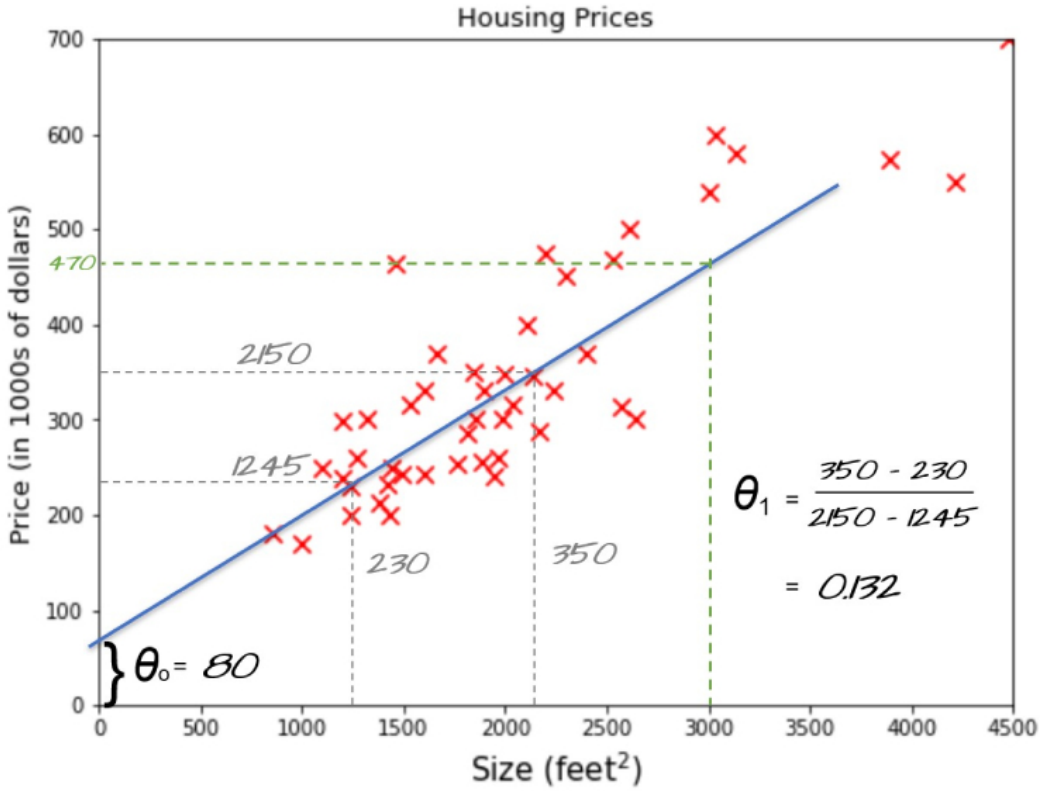
1.1 Introduction

Machine Learning is a branch of AI and Computer Science that uses data as its main input and we humans apply certain algorithms to produce our desired output. In today's world machine learning has a great importance in many things like controlling a self driven robot or predicting some stock price, etc. Machine Learning has two sub divisions - Supervised and Unsupervised Learning. In this chapter we will talk about supervised learning.

Supervised Learning - In this type, we provide our learning model with many inputs(x) and also their corresponding outputs(y), which we treat them as examples. Now applying appropriate algorithms, our learning models tries to learn from the examples provided, to predict an output(y) for a different input not in the examples. For example, if we provide input(x) of a stock to predict its price as output(y), and once the model learns to predict the price of a stock correctly, we can provide it with any kind of input expecting a decently correct output.

Unsupervised Learning - In this type, we provide our learning model with only inputs(x) and no such output(y) is in fact needed. In this type of learning, our main goal isn't to predict anything, but is it actually cluster a group of data according to they similarities or find an abnormality in the data or to make a self driven robot or car to work. These examples will be seen later in the unsupervised learning chapter.

1.2 Linear Regression



Linear regression is easier to understand through this example given above. In the above graph the red crosses indicate the examples or the training set(generally known as) which we provide to the learning model. Here, we are trying to predict the housing prices based on one feature which is the size of the house. So the input (x) here is size of the house (x axis). While the output (y) which we want to predict is on the y axis. Once we provide the examples, we can plot those on the graph as shown above with the red crosses. One common algorithm of predicting these kinds of output is to simple fit the data with a linear straight line also called linear regression. As you can notice, for many examples, the blue line doesn't suit exactly, but out goal is only to approximate the output, it isn't necessary to be exact. This just gives us a rough idea. So according to the line if we want to predict the price of a 2100 sq.feet house, through the graph and the line, we observe that the price could be approximated to nearly 350,000$. Or say 1245 sq.feet, approximate price of house could be around 230,000$. As you can see, linear regression is very useful maybe for property dealers or developers in estimating the correct price. Now how do we come up with that blue line?

1.3 Gradient Descent for linear regression

Now, equation of a straight line can be written as y=w.x+b, where w and b are the parameters we need to find out. To find these parameters, we use gradient descent algorithm.
The overall idea is that, we choose an arbitrary w and b and keep on slowly change w and b, till the sum of the squared error function is minimum. We use this squared error function to get an idea of how far off our line is to the examples or to the training set we have provided to the model.
The squared error function is given by -

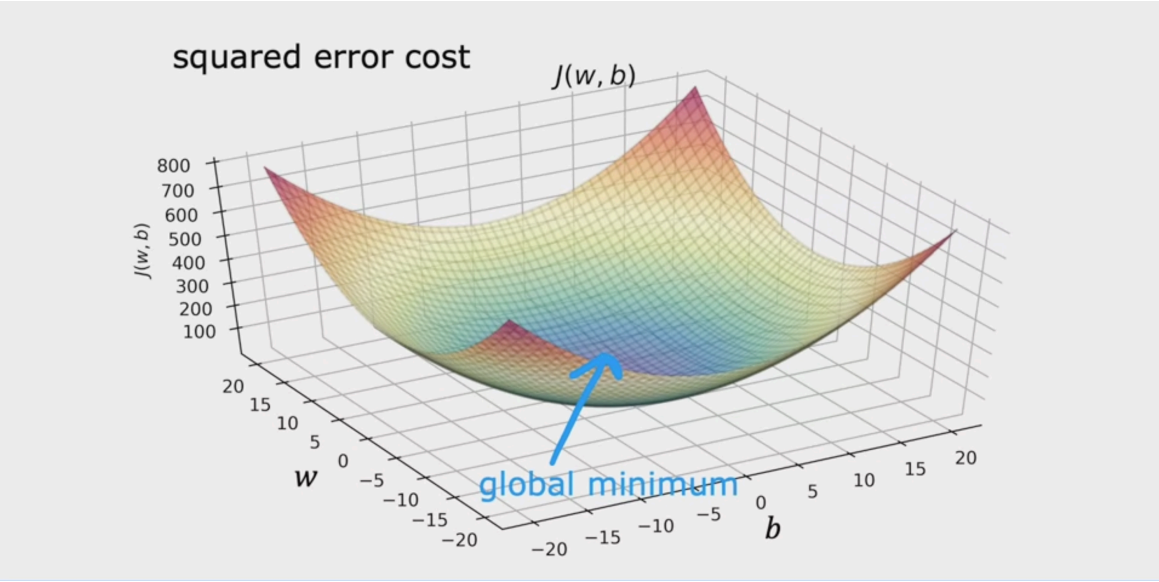
$$\frac{1}{2m}\sum_{i=1}^{m}\left(f\left(x^{(i)}\right)-y^{(i)}\right)^{2}$$

Here the f function is just the straight line equation w.x+b where the letter 'i' denotes the i^{th} training set example. $y^{(i)}$ denotes the real output of the i^{th} training set input. This squared error function is denoted by J (w,b).
The graph of this function looks similar to a 3d bowl and will have a minimum at some value of w and b. But how do we keep changing the value of w and b? For this we have two more similar algorithms applied to the w and b -

$$w = w - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}\left(f\left(x^{(i)}\right)-y^{(i)}\right)\cdot x^{(i)}$$

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

If observed carefully the RHS term of w, the term with alpha in it, actually is the same as (alpha) * (gradient of cost function wrt w). Similarly in the 2nd equation the second term in the RHS is same as (alpha) * (gradient of cost function wrt b). The reason why we are doing this, why are we finding the gradient, in simple words is to reach the global minimum via the fastest route from any random point.
The fastest route here means to go in a particular direction fastest covering less distance which means finding out the gradient itself.



Here if we start at a random point, and apply the 2 formulas given for w and b, and check the cost function as well, what we end up doing is slowly move from the random point towards the global minimum. We stop once the cost gradient wrt w and b both become zero. Now α is called the learning rate and it decides how fast would w and b converge to the corresponding w and b of the minimum cost function. Choosing α wisely is very important and we just can't choose to be big and hope it will reach the minimum point quickly.

The problem is that if you keep the learning rate too small, the model will take a very long time in finding the minimum, which might increase the operational costs significantly. On the other hand if you keep the learning rate high, a bigger problem might occur, the model might not reach the minimum itself, it might deviate from it and the cost function will keep on increasing and so the whole model will make no sense.
Thus to choose a right value of the learning rate, it is necessary to keep a check on how your model is training itself. If the cost function increases, then surely something is wrong, maybe with the algorithm itself or with the learning rate. Or maybe if the model is taking too much time, then the learning rate might be too small, and thus this helps to choose a right α.

1.4 Multiple Regression

Now if we want to predict housing prices, only the size of the house might not be enough or correct. We might need to include other features like number of bedrooms, number of floors or age of the house, etc. How to proceed with this?

Everything remains the same, here the difference arises as the input (x) is now more than one, so instead of taking a single input, we take the multiple features as $x_1,\ x_2, \ldots$ till n, where n are the total number of features. Now the gradient descent algorithm too changes as we need to find $w_1, w_2, \ldots$ corresponding to each of the $x_1,\ x_2, \ldots$

$f\,(x)\ =\ w_1 x_1\ +\ w_2 x_2\ + \ldots$ (till n features) + b

The above function f also looks like this now where x now can be treated as a vector of n features. Same can be said about w, a vector. So f(x) can be rewritten as -

$f\,(x)\ =\ w\,.\,x\ +\ b$, where w.x is the dot product between two vectors.

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

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

Here we can see we need to apply gradient descent for all values of w. Here 'j' corresponds to the j^{th} feature of the input, while 'i' corresponds to the i^{th} example from the training set.

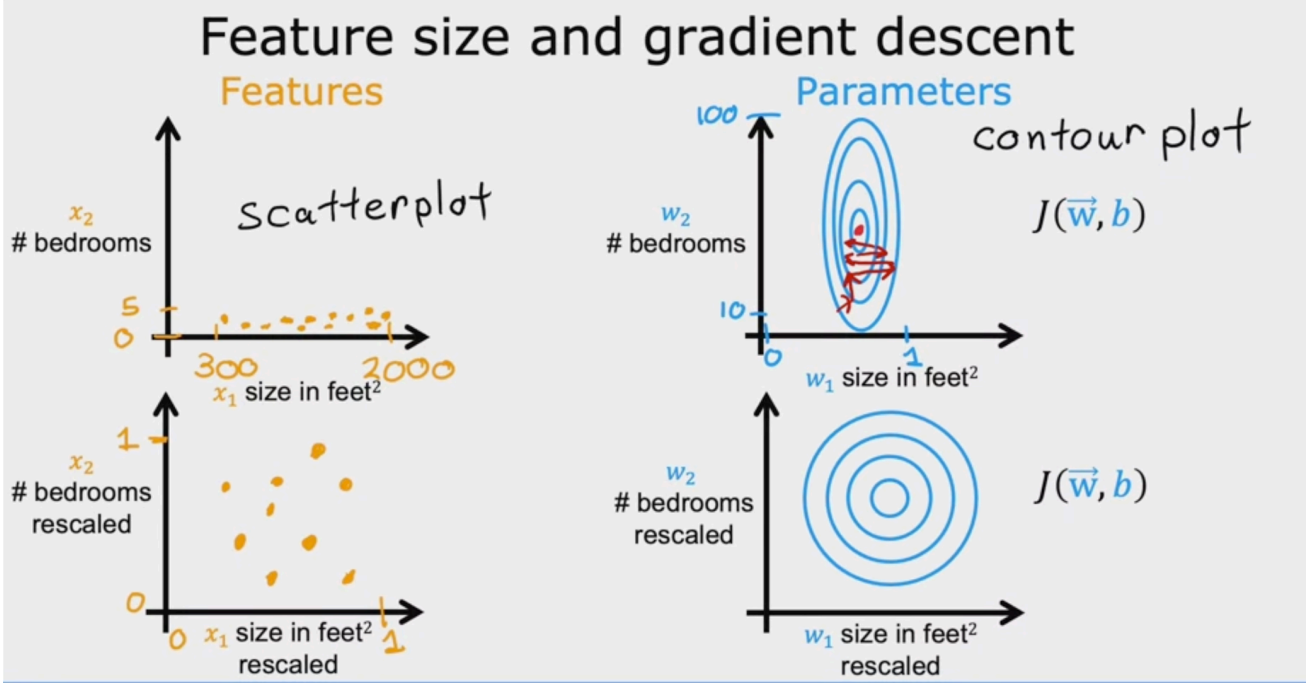
Sometimes a straight line might not be suitable, so we use polynomial regression, in which in f(x) instead of taking the equation as a simple line like w.x+b, we try something like taking $x^2\ or\ x^3$. In case of multiple regression we sometimes take $x_1 x_2\ or\ x_1^2 x_2$, etc. This makes our graph look like a curve which in some cases be more useful in approximating, but overall the gradient descent algorithm remains the same.

One more important part of the algorithm is feature scaling which is useful in making the algorithm work better. In many cases the values of w (in multiple regression) might be very different for each input feature. Also while minimising the cost function, we also can use contour plots between w and x. But there might be a big difference between the input features like x_1 = 4 and x_2 = 1000, and this fully depends on what kind of feature we take. To make the contour plot more visible and to make it more observable in graphs, we can rescale the features.

The most common method is to use this formula -

$$x_j = \frac{\left(x_j - u_j\right)}{\max\left(x_j\right) - \min\left(x_j\right)}$$

Here u_j = mean of $x_j's$, where j is the input feature.
In this way all features can be in between 0 and 1.



Here we see the above two graphs before feature scaling, everything looks crumbled, but the below two graphs are after feature scaling. This is an example of predicting house prices.

1.5 Logistic Regression

Just like linear regression, in logistic regression we also want to predict something. The main difference here is that instead of predicting some integer value like price of a house or stock price, we just want a answer in yes or no.
For example, if we want to know if a tumour found in a human body is cancer threatening or not, just want answer in yes or no. In this case we work it out just like linear regression and get an output as 1 or 0. Here 1 represents yes and 0 represents no.
What we try to do is set a boundary condition, where if the input we give crosses the boundary, we get an output as 1 or 0 depending on what kind of condition we set.

1.6 Gradient Descent for logistic regression

The cost function or the function we want to minimise is the same as in linear regression -

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

We can directly jump to multiple input features. Here x is in vector form.
But f(x) in logistic regression is very different. Instead of a straight line or a simple polynomial, it is rather a sigmoid function -

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

Here w and x can be considered as a vector, be it with 1 element or more than one. So we have defined logistic regression directly with multiple input features, which is pretty much same as with one input feature.
There is another way of representing the cost function -

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

The gradient descent algorithm remains the same -

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

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

As in linear regression, we keep on updating w and b simultaneously till we reach the minimum of the cost function.
We can also use concepts like about feature scaling, learning rate, etc. in logistic regression. Overall the algorithm remains the same.

1.7 Overfitting

Overfitting occurs when your learning model has trained itself more than needed on the training set, which just means the model was just made to fit the learning model, and not learn the trend. As a result what might happen is even though it worked perfectly on the training set, it might fail on any random example different from what we provided to the model.
Ways to prevent this -

1) Carefully select which input features to include, do not include unnecessary features not needed.
2) Collect more data (training set)
3) Regularisation

Regularisation -

Do not make w_j too big, try to make them relatively small, so that any input feature does not get unwanted priority while predicting something. What this will do is make only one or two features of utmost importance as they might contribute a lot in the cost function. So we generally add a regularisation term -

$$\frac{1}{2m} \sum_{i=1}^{m} \left(f\left(x^{(i)}\right) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_j^2 + \frac{\lambda}{2m} b^2$$

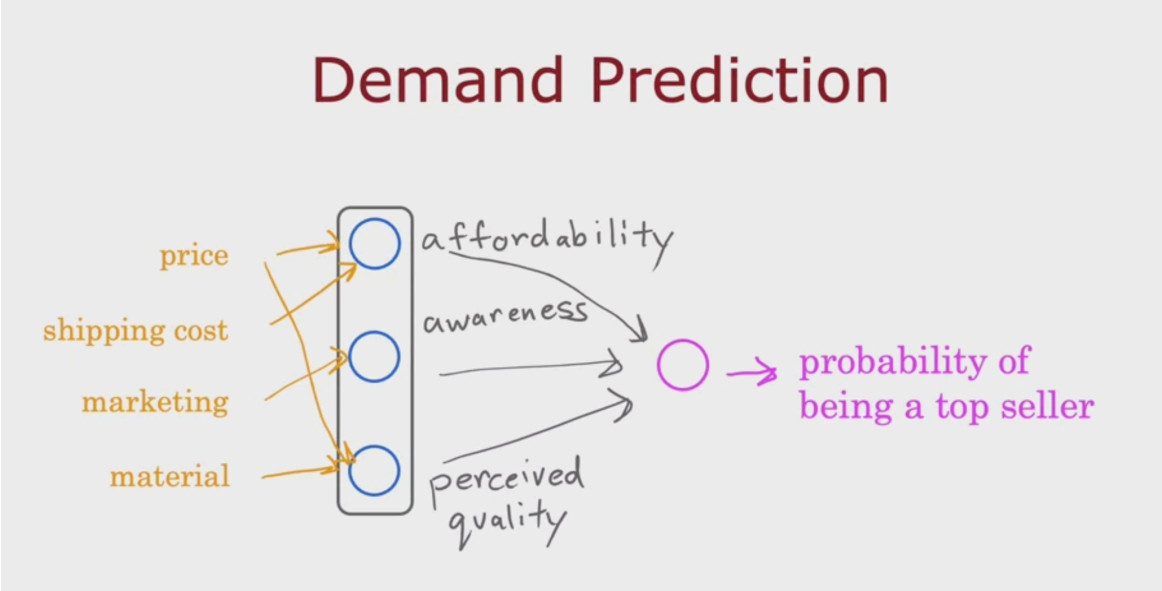
This cost function helps the gradient descent algorithm to make a particular w_j to be very big, same with b. Values of λ will be discussed later.

CHAPTER 2

Advanced Learning Algorithms

2.1 Introduction to neural networks

The main motivation to learn or implement neural networks is to try to imitate the function of the neurons in human brain. What the neurons do is to to take as input information on whatever we see or hear or feel and pass these information to other neurons through certain functions. These functions are similar to what we saw in linear and logistic regression. Once this information is passed, it is processed thoroughly until this whole system of neuron transmission ends with a single output neuron.



In the above picture we see input as 4 features (price, shipping cost, marketing and material). These 4 features are used to predict more 3 new features(affordability, awareness and perceived quality) by functions called as activation functions. These 3 new features together forms a neuron, which then is used to predict the probability of maybe a shirt of being a top seller as the final output neuron. This is how neural networks work, and we think our human brain work. Similarly there might be a huge neural network working, the one which we saw above is just a model with a single layer.

2.2 Activation functions

Taking the above example, the one layer in between is very important and basically the deciding factor. If we see the picture carefully, we can see yellow arrows pointing from the input to the features in the neuron layer. This just means that, say to predict affordability, the input feature we need is the price and shipping cost as shown. Similarly for awareness, we need only marketing.
Now according to the activation function we use for the layer, we use that same function in the cost function we learnt in linear regression and apply the gradient descent algorithm to train the model.

Let us assume the layer one is labelled a linear function(input to layer 1) and layer 1 to output as sigmoid function. Now affordability can be written as w.x+b, since layer 1 should be computed by linear function. Here x is a vector containing price and shipping cost explained in first paragraph. Now with training the model with gradient descent, w and b would be found out. Let us make a notation - $w_j^{[i]}$, where i denotes the i^{th} layer and j denotes the j^{th} neuron in the i^{th} layer (same for b as well). So for affordability we can write the parameters as $w_1^{[1]}\ and\ b_1^{[1]}$. Similarly the model will find the remaining parameters for awareness and perceived quality with the appropriate input features. Now whatever output this gives is a vector with 3 elements which can be written as $a^{[1]}$ with $elements\ a_1^{[1]}, a_2^{[1]}, a_3^{[1]}$.

For the final output since the activation function we assumed is sigmoid function, we take as input $a^{[1]}$ vector and find out the necessary parameters with gradient descent. So finally our output would be written as - $a^{[2]} = g\left(w_1^{[2]}.\ a^{[1]} + b_1^{[2]}\right)$, where g is the sigmoid function.

So this is how neural network is trained and how it actually works. We can use this model in a variety of ways to predict more than one single quantity also.

One more activation function is the ReLU defined as -

g(z) = 0 ; z<=0

g(z) = z ; z>0

Generally sigmoid function is used in the middle layers and in the output layer, if the output can be negative as well, then we generally use linear function, but if the output can only be positive, we prefer the ReLU function. Sigmoid can be used if we want a value between 0 and 1 like maybe probability.

2.3 Multiclass classification with some more concepts

Multiclass classification just means classification or prediction between more than 2 things, maybe 3 or 4. Like in logistic regression, we predict a binary output like yes or no. In multiclass classification, we might have to predict maybe more than just a binary output, maybe 4 or 5 but surely a finite number. For example, if we want to predict probability of say 4 events.
This is called softmax regression.'

In this we find out all values of the particular neuron in one layer namely $z_1, z_2, \ldots, z_n$ and to find $a^{[j]}$ for the j layer, we use this formula for the final output neuron -

$a_i^{[j]} = \frac{e^{z_i}}{(e^{z_1} + e^{z_2} + \ldots e^{z_n})}$ and find out the probabilities or whatever we want to.

One advanced optimisation of the gradient descent is the Adam algorithm. While we are applying gradient descent, what this algorithm does is changes the value of the learning rate α continuously depending on how well the model is performing or proceeding with minimising the cost function, it can decrease the learning rate if it finds the cost function increased at a point or increase the learning rate if it finds that the cost function is decreasing very slowly.
If we have 8 values of w and one value of b in the whole model, in Adam algorithm while applying descent algorithm there is not a single global α but

instead we have to keep 8 values $\alpha_1, \alpha_2, \ldots \alpha_8$ all different for the 8 values of w, which in a way proceeds with the gradient descent differently according to the needs. This optimises and makes the model better, it also helps to train the model faster and hence is very useful.

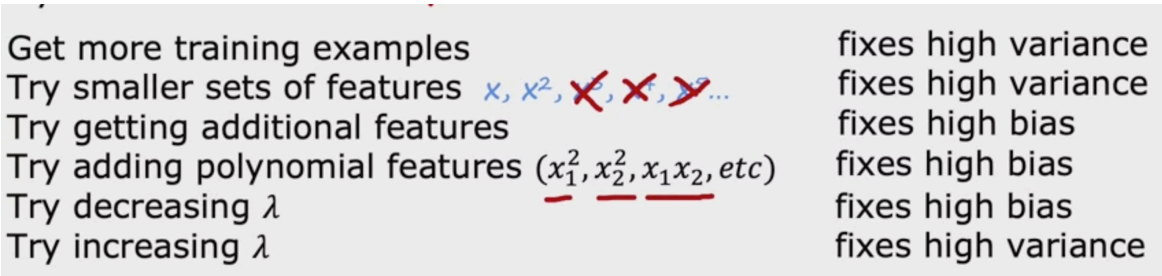
One more optimising way is to train the model with only maybe 80% of the training set, keep 10% for cross validation or checking if your model is working right or not, and keep remaining 10% for test cases. This prevents overfitting of your model.

2.4 Bias and Variance

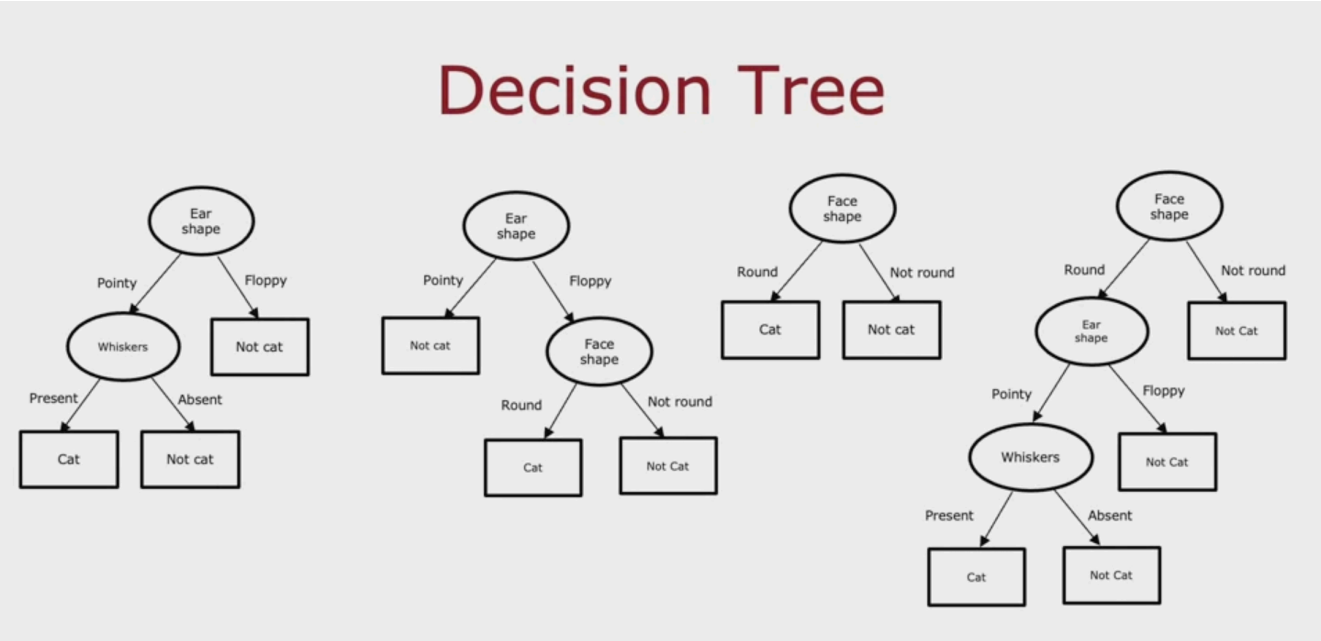
High bias simple means under-fitting the data, meaning the model is performing poorly on the training set itself and it yet hasn't been ready on the examples provided. High variance is a case of overfitting your data.

By dividing a little of the training set into cross validation and test cases, if our cross validation error is very much compared to the training set, that is a case of overfitting, i.e. high variance. While if the cross validation error and training set error is almost same but the training set error is larger compared to what it was predicted to be, this means this is a case of under-fitting, i.e. high bias.

Talking about λ, it can be seen that if we keep it's value high in the cost function given in the regularisation section, then all values of w would want to be minimised which might under-fit the data. While if we keep λ very small, we can just ignore those terms, and there is a chance of overfitting the data. So a way to deal with this is to compute the cost function for several values of λ, starting from very small continuing up to a large value and then choosing the one with lowest cost function. Below is a way to fix bias and variance -



2.5 Decision Trees



The above picture contains 4 decision trees, but it is very clear from the picture, what is a decision tree. It is a type of flowchart, where at each point in the flowchart, based on a feature, the flowchart gets divided just like how it happens in logistic regression.

How to choose what feature to split on?

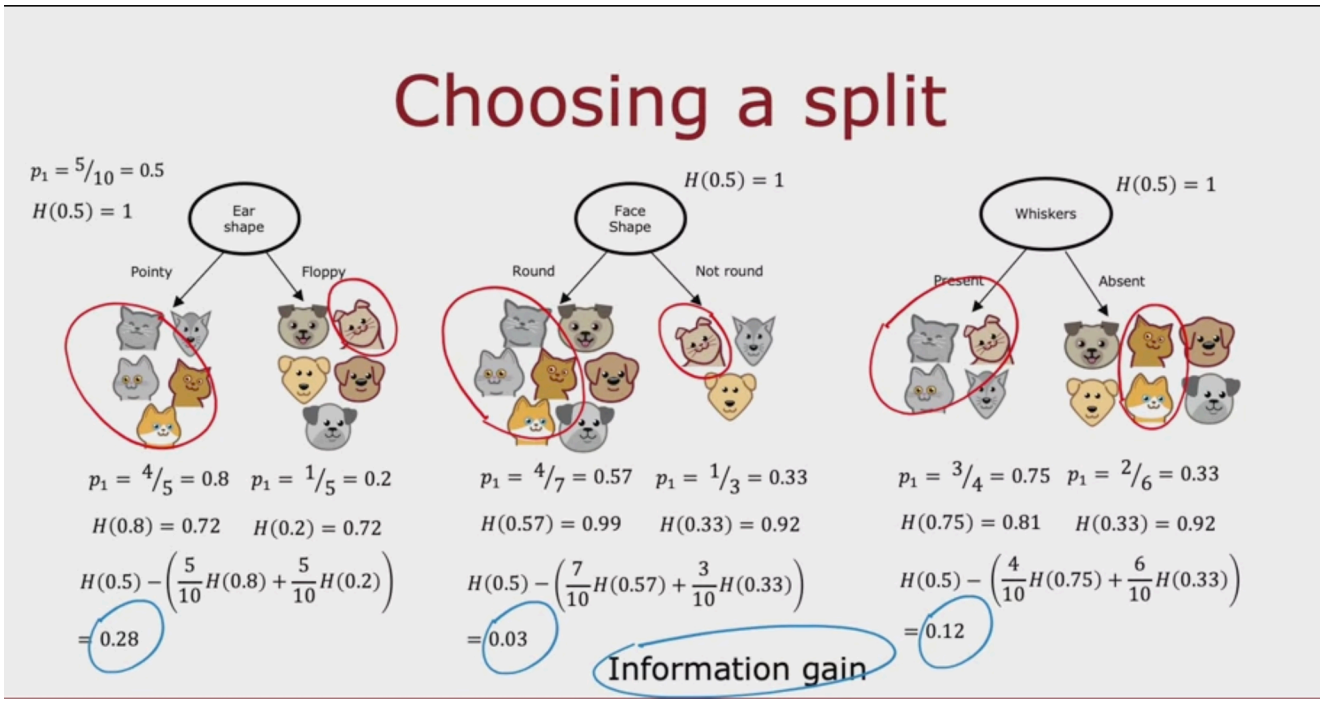
There is a term called entropy, and our aim is to minimise entropy or maximise the purity of the output. Purity means how much quantity in the output belongs to one type. Now say we get output of 4 cats and 1 dog, but we want to separate out cats and dogs. So the purity of output is 4/(4+1) *100 =80%, if there were 5 cats and 0 dogs, the purity would be 100% since all belong to one category (cats).

First we need to find p_1 which is the fraction of finding a cat in the example, just as shown in last paragraph as 4/5. Now entropy of that is -

$$H(p_1) = -p_1 \log_2 (p_1) - (1 - p_1) \log_2 (1 - p_1)$$

This looks very similar to the term inside the summation part while computing cost function in logistic regression.

But we won't just compare the entropy, in fact we have to find a kind of weighted average and compare it between all possible decision trees possible and select the most appropriate one.



Here the difference between H(0.5) and the weighted average of the entropies of LHS and RHS side gives us what is called the entropy reduction. What we want is entropy or the randomness to be less or ideally 0. So for getting a better decision tree, we want the entropy to be reduced the max, and so the entropy reduction should be more. In this way we check layer by layer and come up with a decision tree. We need to continue doing so until the following criteria is met -

1) When entropy becomes 0
2) When the number of layers in the decision trees or number of divisions exceed a particular number
3) The information gain or entropy reduction reaches a certain minimum threshold
4) When the number of examples after division is itself below a certain threshold

Also decision trees can also be used to predict more than just classifications based on binary input. Like for classifying cats and dogs, decision trees can even take weight as input. To do so we need to subtract the the weighted average of the variance of weight of the LHS and RHS after division, with the variance of weight at the top most layer of decision tree. Now compare which value is the greatest. That corresponding division is to be made.

2.6 Tree Ensemble

Making a single decision tree does not work since even a small change to the training set can completely change the whole decision tree itself. So the main idea is to create multiple decision trees, generally a lot of them, and try changing the training set by adding a new fresh example and put it in each decision tree. Our goal is to train the model and pick the best decision tree with the least error.

If we don't have enough data available, one way of creating new examples is by sampling with replacement. But every time we apply this algorithm, it is best if we give more probability to the example which the previous decision tree had misclassified.

One more algorithm called the random forest algorithm is to choose the feature we want to split based on from a randomly chosen subset of features only. Do this every time with all the decision trees.

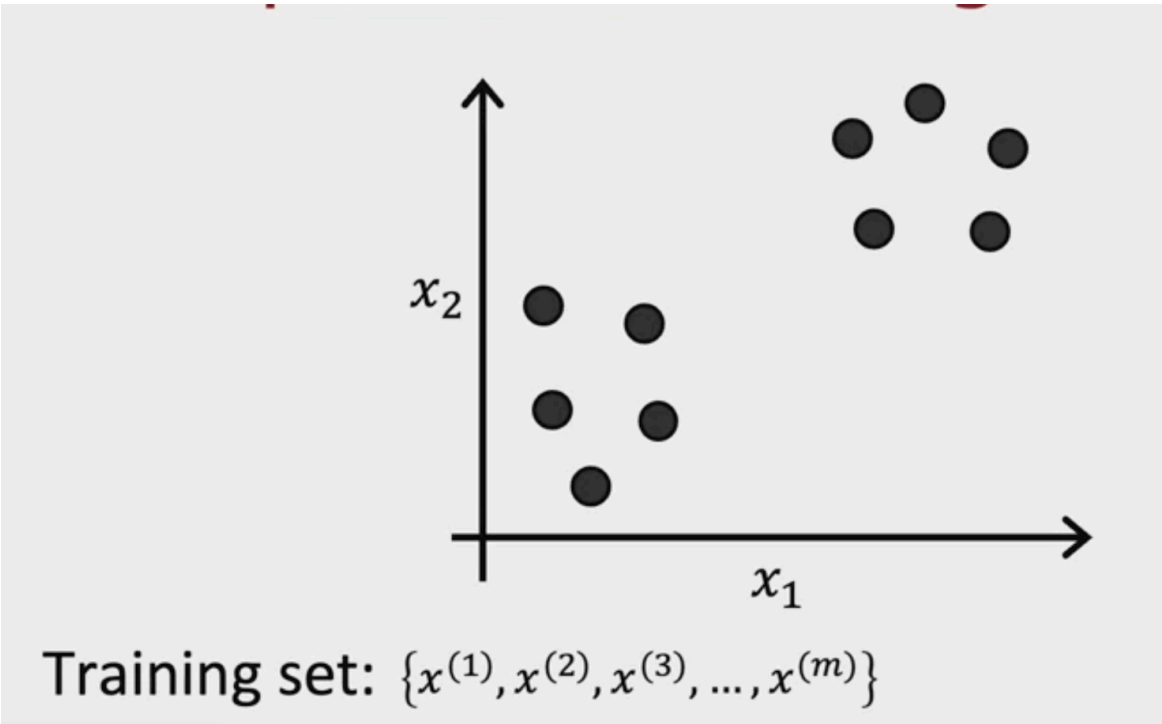
Decision trees work well on structured data and not on images, audio or text, like identifying a voice or identifying an image, etc. Also humans can intervene in small decision trees which would bring out the best possible result.
Neural network must be used for a larger set of data, might be structured or unstructured. It might run slower than a decision tree, but you can use a neural network built by someone else in your work as well which won't be the case in decision trees.
Both of them are useful but we need to be wise in choosing which one to work with.

CHAPTER 3

Unsupervised Learning : Recommenders and Reinforcement Learning

3.1 Clustering

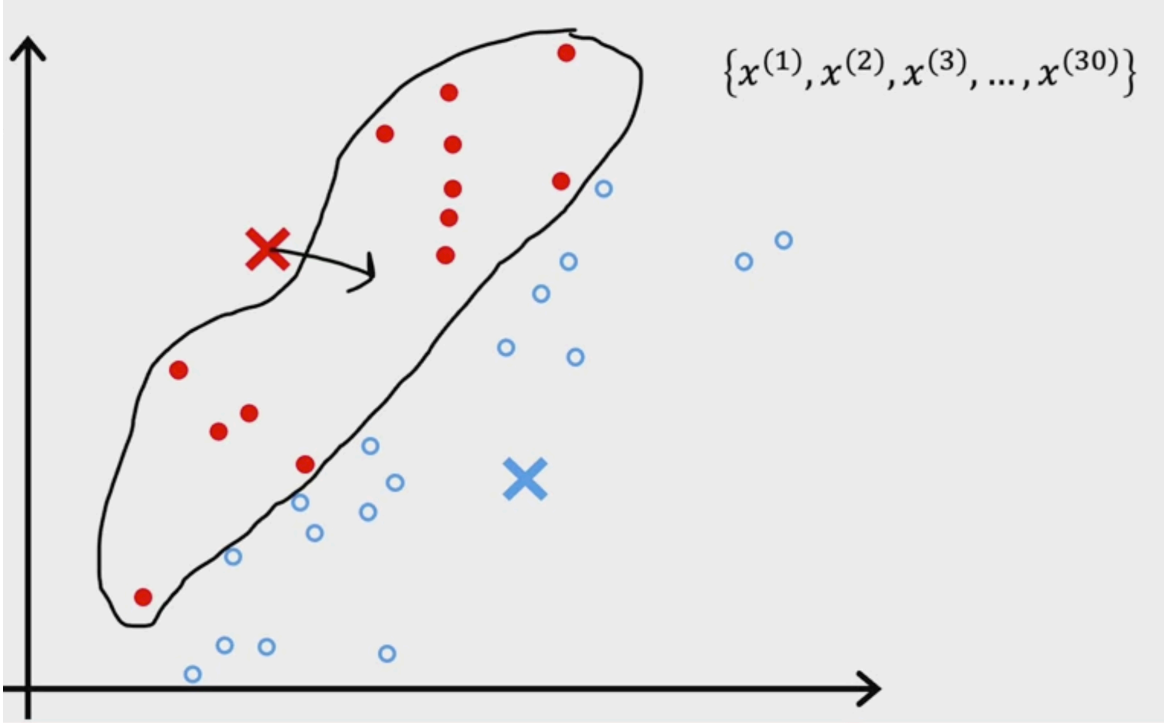
Clustering is like training a model on a certain inputs (x) with no output but to only find clusters or groups between the inputs.



In the above picture, we have say 10 examples from the training set. We can see imminently that 2 clusters or groups are made based on input features x_1, x_2. Now we can form 2 different groups called clustering.

The algorithm to do this is as follows (K means algorithm) -
Suppose we want to find K clusters. Now we randomly initialise K points on the graph anywhere. This K points are called the cluster centres and our aim is to shift these K points to their corresponding cluster centres. Now find the

distance between K points and each of the example from the training set. Assign the points from the training set to one of the K point whose distance is closest to the point. Now whatever points are associated to one of the K points, now shift the same cluster centre to the point on graph which is the average of all the point assigned to it in the previous step. Continue doing this till the cluster centre does not shift anymore.



As in the above picture, the red dots are assigned to the red cross since all the red dots are closer to the red cross than the blue cross. Similarly for the blue cross. Then taking average of all red dots and blue dots gives the new position for the red cross and blue cross respectively. In this way we form groups in the data.
Also generally there is no perfect way of selecting number of clusters for a data beforehand. The best way is to select the number of clusters you want based on the purpose of making the model.

Clustering is used in grouping similar news together, or for DNA analysis, or maybe for deciding T-shirt size based on weight of appropriate customer and size of the shirt itself.

3.2 Anomaly Detection

In this, our main goal is to find some abnormality in the data or some odd data which doesn't match with other input examples. We use this in many application like picking out faulty engines manufactured based on many input features.
What we do is find final output probability and fix a threshold above or below which the input is considered to be an anomaly.

$$p(x) = \prod_{j=1}^{n} p\left(x_j\ ;\ \mu_j\ ,\ \sigma_j\right)$$

where, $p\left(x_j\ ;\ \mu_j\ ,\ \sigma_j\right) = \frac{1}{\sigma_j\sqrt{2\pi}} e^{\left(-\frac{\left(x_j-\mu_j\right)^2}{2\sigma_j^2}\right)}$

The above term just shows the the p function as a gaussian distribution. Here, σ denotes the standard deviation of the training set based on feature x and μ denotes the mean of the training set based on feature x. Now we set ϵ to a value such that p(x) < ϵ will denote anomaly. So this means the probability of it becoming normal is very less, no an anomaly.
In this also, we separate some of the training set examples for cross validation cases and test case.

If we see the graph between the training set example and feature x is not a gaussian distribution type, we can change the input feature a little bit like maybe to log(x) or x^2 so that graph looks like a gaussian distribution graph.

Also we can merge 2 or more features into one itself if we see the value of one of them is too large or too small or making a gaussian distribution graph out of it is hard. Merging two examples could be done like x_1x_2 or x_1/x_2.

3.3 Collaborative Filtering

There is something called recommender system which recommends the users maybe what to buy on online websites or recommends users movies to

watch. Let's take an example of a movie recommender system. It uses collaborative filtering algorithm.
What it does is tries to give a value to the movie based on its feature. Like if the movie is a horror one, then out of 5, horror feature gets 5, while comedy feature gets 0, basically an overview of the movie. Now according to the users previously based rated movies, the system tries to find w and b parameters and apply regression algorithm here.
Once it finds these parameters whatever rating output is predicted, if it above a threshold, that movie is recommended. But the overview of the movie also sometimes depends on the user.

$$J\left(w,b,x\right) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(w^{(j)}.x^{(i)}+b^{(j)}-y^{(i,j)}\right)^2+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(w_k^{(j)}\right)^2+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_k^{(i)}\right)^2$$

Here $r\left(i,j\right)$ represents if the user j has given rating on movie i or not. It's value is 1 if rating is given and 0 if not given. Other all parameters are same like in earlier cost function. Just that n_m= number of movies, n_u= number of users, and n = number of movies the user has given rating.

Now the system applies gradient descent algorithm normally, just that it applies the algorithm not only to w and b, but also x now as it itself is a parameter if the system does not know what kind of a movie is it. This whole process also works if there is no rating and only if it has binary option to rate a movie, maybe yes or no. In that case simply work with binary inputs like 0 and 1.

Finding related movies is simple and we just need to compare the input features and minimise the distance between them - $\sum_{i=1}^{n}\left(x_i^{(k)}-x_i^{(l)}\right)^2$

The smaller the value of this the more similar is the movie. Here movie k would be similar to movie l.
There is one small algorithm known as mean normalisation also, in which we first take the average of the rating of a movie and then subtract it from each user's rating they had given to the movie. Finally we will add that value to the prediction which is w.x+b. This optimises the recommender system even more.

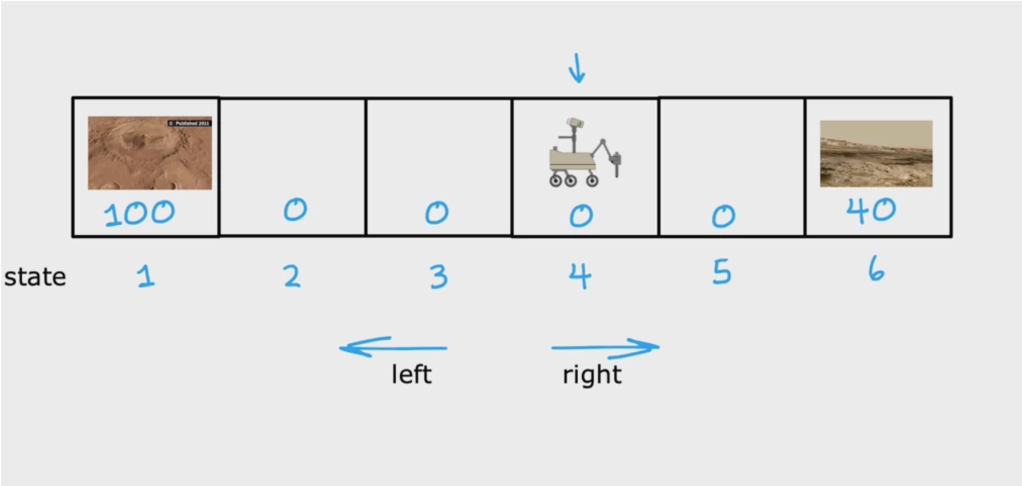
3.4 Contest-Based Filtering

This is one more way of recommending movies to the users. It recommends movie based on the features of the movie, i.e. average rating of the movie, how old the movie is, etc. but also takes in account the feature of the users like age of the users, where do they live, gender, etc.

This works out just like collaborative filtering. After we have got a whole bunch of features of users (x_u) and features of the movie (x_m), our goal is to minimise all these features into two vectors v_u and v_m of equal number of elements respectively for user feature and movie feature. The way to do this operation is by supplying the bunch of features we collected in a neural network. We generally use a single neural network for both x_u and x_m.

Generally the neural networks to do this task is already present on internet and we do not have to make a whole neural network from start. We can make changes to it according to our needs. Also if we want to train the neural network, then the f(x) in the cost function would be only the dot product between v_u and v_m. We ignore the b term. Other than that the regularisation terms remain the same(the term containing λ).
Tricks like finding 10 most similar movies according to user's recent watched movies, top 20 movies of the user's country can also help in making the neural network stronger and can produce better results.

3.5 State-Action Value function



In above picture, we see an example of a rover trying to reach different states or say places. It is currently in state 4. The values written the bottom of each state is a price given to it. If the rover reaches a state, it is awarded with that many points say (points = value written at the bottom of each state). Our goal is to find the most optimum path for the rover so that it gets maximum number of points.

There is a term called discount factor (γ). Also we calculate the return which gives us an idea of which path to take. Let's us say the rover took 2 paths, state 4 -5-6 and other one as 4-3-2-1. The return is calculated as -

$R_1 + \Upsilon \,.\, R_2 + \Upsilon^2 \,.\, R_3 + \Upsilon^3 R_4 + \ldots$,where $R_1, R_2, \ldots$ are the price awarded to the rover to be in that state. Also starting from initial point, these rewards are to be used in the formula sequentially as it passes from say 4 then to 5 then to 6. Now keeping γ=0.9, and according to above picture, the return is as follows -

For 4-5-6 path, return = 32.4

For 4-3-2-1 path, return = 72.9

Clearly 4-3-2-1 path is better. We use discount factor and change it according to our needs. Like if we want to reach our destination as fast as possible we might increase the discount factor.

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

In above we can see two values written in each state. The RHS value represents the return value of going from that state and continue towards the right, while the LHS value represents the return value if the rover continues towards left.

The above equation is the Bellman's equation and s stands for current state and a stands for action the rover will take from the current state s. R(s) is the reward the rover will get in state s, while s' and a' represent the next state and action after reaching state s and taking action a. This whole process needs a neural network which can make the work easier.
This formula is the algorithm needed into a reinforcement learning model to make the rover go optimally.
All the values shown in last picture is calculated only after the rover is found to be in such a situation by the bellman equation. This way state action value function Q(s,a) works via neural network.

3.6 Continuous state spaces

Instead of the next action only decided by the initial state and action taken before, we need to have information of the speed of the rover in all directions, maybe also the angle with which is inclined to the horizontal, etc. We can define the rewards of all actions maybe a very high reward for a perfect landing and high negative reward for a crash landing. Similarly for moving up and down, we need main engine boost which we could give a small negative reward. For moving left and right, we can again a give a small negative reward for side boosters.
Here also we use the Bellman equation with the new types of state and action and rewards. We could create a neural network by taking a specific state and 10000 such training examples. Then we could train it using neural network in detail using bellman equation and at the end we build a neural network which on repeated learning gives the best result.
One way of choosing an action while still learning is to pick the the action that maximises Q(s,a) with probability 0.95 and remaining 0.05 probability of choosing a random action a. This helps in exploring of all action even more and could also help reduce the time taken for the neural network to be trained. This is called the ϵ-greedy policy. Here ϵ = 0.05. Generally we keep the value of ϵ high close to 1 and gradually decrease it over time. The only problem with reinforcement learning is that it has very few applications compared to supervised learning, but is very fast and easy to apply in real life situations.