

Topic - 03

Training Models

Part: 01



+88 0172 6867 984



tanvir@socian.ai

Introduction

From the last few lectures and labs, you might be actually wondering that Artificial Intelligence or Machine Learning is very easy after all. You didn't have to see how the classifiers work. So far we have done:

- LifeSatisfactionPrediction
- Sentiment Analysis
- Win/Loss Prediction
- Handwritten Digit Recognition
- Facial Recognition of People

And we have used many Regression and Classifiers/Algorithms. But we haven't looked under the hood. Indeed, in many situations you don't really need to know the implementation details. But...

Introduction

Having a good understanding of how things work can help you –

- Quickly come up with the appropriate model
- The right training algorithm to use
- A good set of hyperparameters for your task
- Debug issues and perform error analysis more efficiently.
- Finally, most of the materials discussed in this topic will be highly essential in understanding, building, and training Neural Networks; which we will need in every step of working on Deep Neural Networks (Deep Learning)

Don't be Afraid of Math and Codes

In this Chapter, we will be looking at lots of Equations of Math as we will see in depth how the things work in real life, not just only the end user output.

Moreover, there will be lots of codes involved in Class Lecture as well.

Please don't be afraid. On the first day of the class, I have told you all that AI is all about Probability and we will see those probabilities in real life.

This is the journey of Awesome Learnings.

Linear Regression

Earlier in our Lab, we have seen how the Life Satisfaction changes over the GDP Per Capita. This was actually a Simple Regression Problem. The Equation we obtained:

$$\text{Life_Satisfaction} = \theta_0 + \theta_1 \times \text{GDP_Per_Capita}$$

This model is just a linear function of the input feature GDP_Per_Capita. θ_0 and θ_1 are the model's parameters.

More generally, a linear model makes a prediction by simply computing a – Weighted Sum of the Input Features, Plus a constant called the Bias Term, θ_0 (also called the intercept term).

Let's look at the equation -

Linear Regression – Cont.

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

Here –

- \hat{y} is the predicted value.
- n is the number of features.
- x_i is the i th feature value.
- θ_j is the j th model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \dots, \theta_n$).

If we look into the vectorized form of this equation, then we find the following:

Linear Regression – Cont..

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

Here –

- θ is the model's *parameter vector*, containing the ***Bias Term*** θ_0 and the feature weights θ_1 to θ_n
- θ^T is the transpose of θ (a row vector instead of a column vector)
- \mathbf{x} is the instance's *feature vector*, containing x_0 to x_n , with x_0 always equal to 1
- $\theta^T \cdot \mathbf{x}$ is the dot product of θ^T and \mathbf{x}
- h_{θ} is the hypothesis function, using the model parameters θ

Mean Square Error (MSE) of Linear Regression

In our previous lectures in Topic 2, we have seen the Root Mean Square Error (RMSE). So, in order to train a Linear Regression model, we need to find the value of θ that minimizes the RMSE.

In reality, it is easier to minimize the **Mean Square Error** (MSE) than the RMSE and ultimately it gives us the same result (because the value that minimizes a function also minimizes its square root)

The MSE of a Linear Regression hypothesis h_{θ} on a training set \mathbf{X} is calculated from the following equation (We will use $\text{MSE}(\theta)$ to keep things simple):

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

Normal Equation of Linear Regression

To find the value of θ that minimizes the cost function, there is a mathematical equation that gives the result directly. The ***Normal Equation***, .

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

Here,

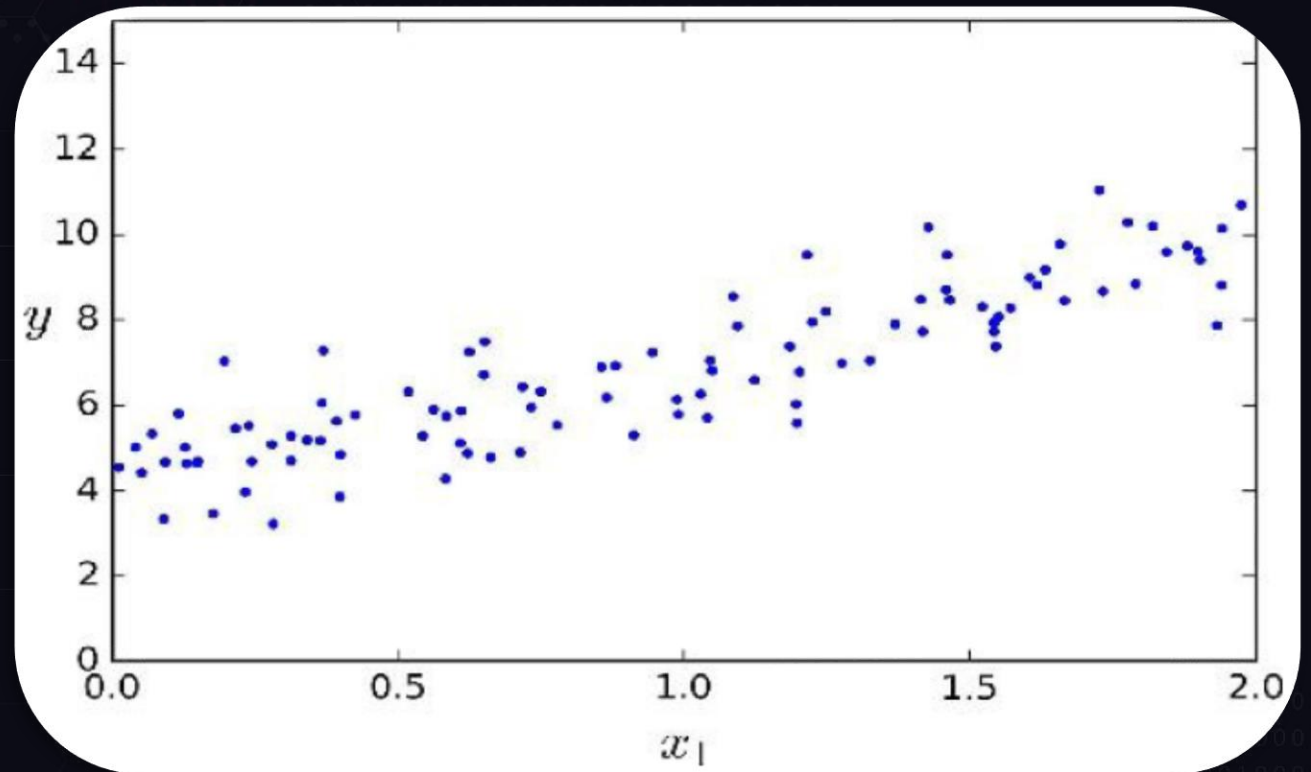
^

- $\hat{\theta}$ is the value of θ that minimizes the cost function
- y is the vector of target values containing $y(1)$ to $y(m)$
- X^T is the transpose of the Matrix X

Now, it's coding time!

Normal Equation of Linear Regression – Coding

```
import numpy as np
import matplotlib.pyplot as plt
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```



Normal Equation of Linear Regression – Coding

Now let's compute the θ^* using the Normal Equation. We will use the `inv()` function from NumPy's Linear Algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

```
from sklearn.linear_model import LinearRegression
# add x0 = 1 to each instance
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
print(theta_best)
```

The actual function that we used to generate the data is $y = 4 + 3x +$ Gaussian noise. Let's see what the equation found:

This is the output I got in PyCharm (Your output may vary):

```
[[3.85414135]
 [3.13070218]]
```

Normal Equation of Linear Regression – Coding Cont.

We would have hoped for $\theta_0 = 4$ and $\theta_1 = 3$
Instead of $\theta_0 = 3.85414135$ and $\theta_1 = 3.13070218$.

This is close enough, but the noise made it impossible to recover the exact parameters of the original function.

Now we can make predictions using :

```
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]
# add x0 = 1 to each instance
y_predict = X_new_b.dot(theta_best)
print(y_predict)
```

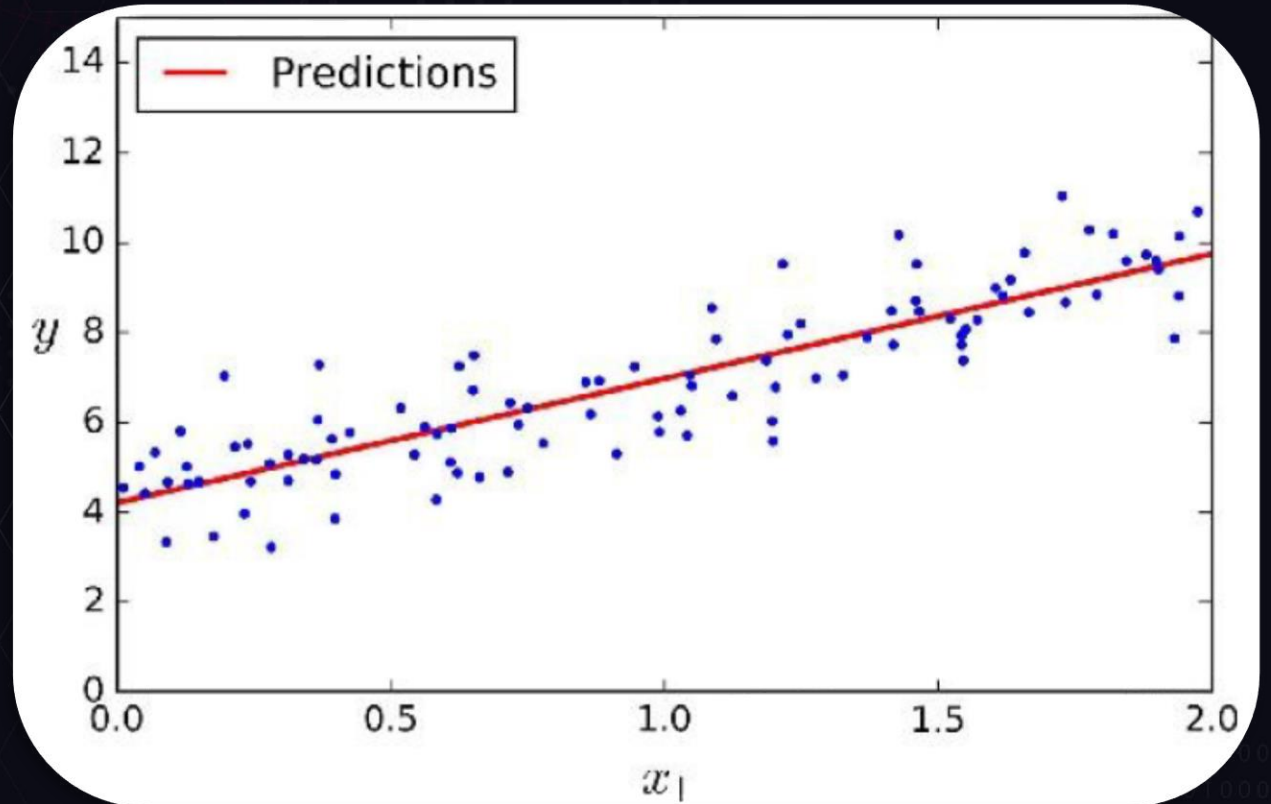
Output (Your output may vary):

```
[[4.05957622]
 [9.94134315]]
```


Normal Equation of Linear Regression – Coding Cont..

Let's plot this model's predictions:

```
plt.plot(X_new, y_predict, "r-")  
plt.plot(X, y, "b.")  
plt.axis([0, 2, 0, 15])  
plt.show()
```



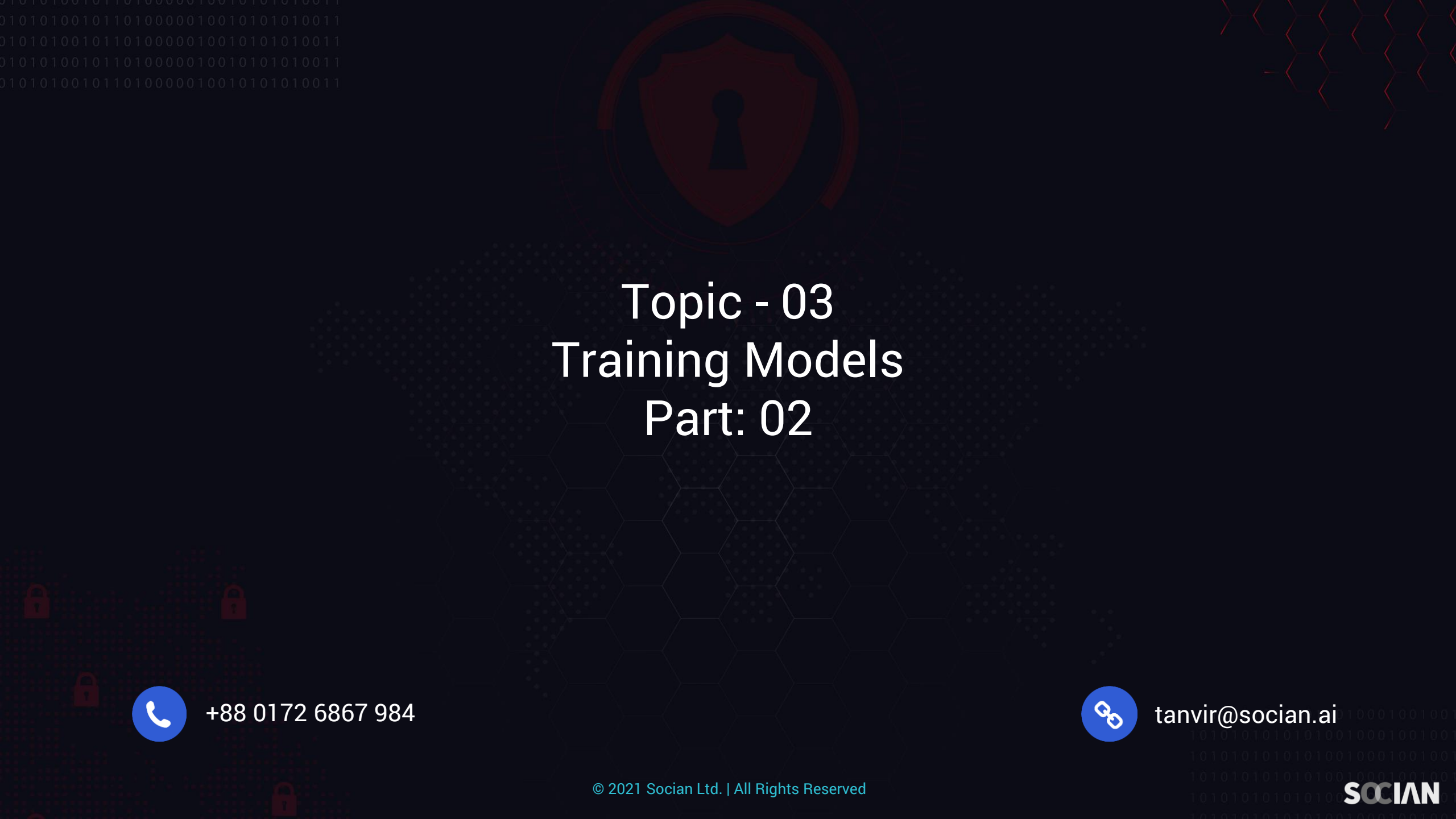
Normal Equation of Linear Regression – Using Scikit Learn

If we use Scikit Learn, the equivalent code looks like this:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
print(lin_reg.intercept_)
print(lin_reg.coef_)
print(lin_reg.predict(X_new))
```

The output that we get would be like the following:

```
[4.04448375]
[[3.16079029]]
[[ 4.04448375] [10.36606432]]
```



Topic - 03

Training Models

Part: 02



+88 0172 6867 984



tanvir@socian.ai

Computational Complexity

The Normal Equation computes the inverse of $\mathbf{X}^T \cdot \mathbf{X}$, which is an $n \times n$ matrix (where n is the number of features). The *computational complexity* of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$ (depending on the implementation).

In other words, if you double the number of features, you multiply the computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$.

The Normal Equation gets very slow when the number of features grows large (e.g., 100,000).

On the positive side, this equation is linear with regards to the number of instances in the training set (it is $O(m)$), so it handles large training sets efficiently, provided they can fit in memory.

Computational Complexity – Cont.

Also, once you have trained your Linear Regression model (using the Normal Equation or any other algorithm), predictions are very fast: the computational complexity is linear with regards to both the number of instances you want to make predictions on and the number of features.

In other words, making predictions on twice as many instances (or twice as many features) will just take roughly twice as much time.

Gradient Descent

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

This is how it works in human life:

Let's say, you were driving by plane and your plane crashed on a roof top of the Antarctica. Now, you are the only passenger who survived. But there was a big foggy snowfall going on in the top of the mountain and hardly only 2 meters from you can be seen.

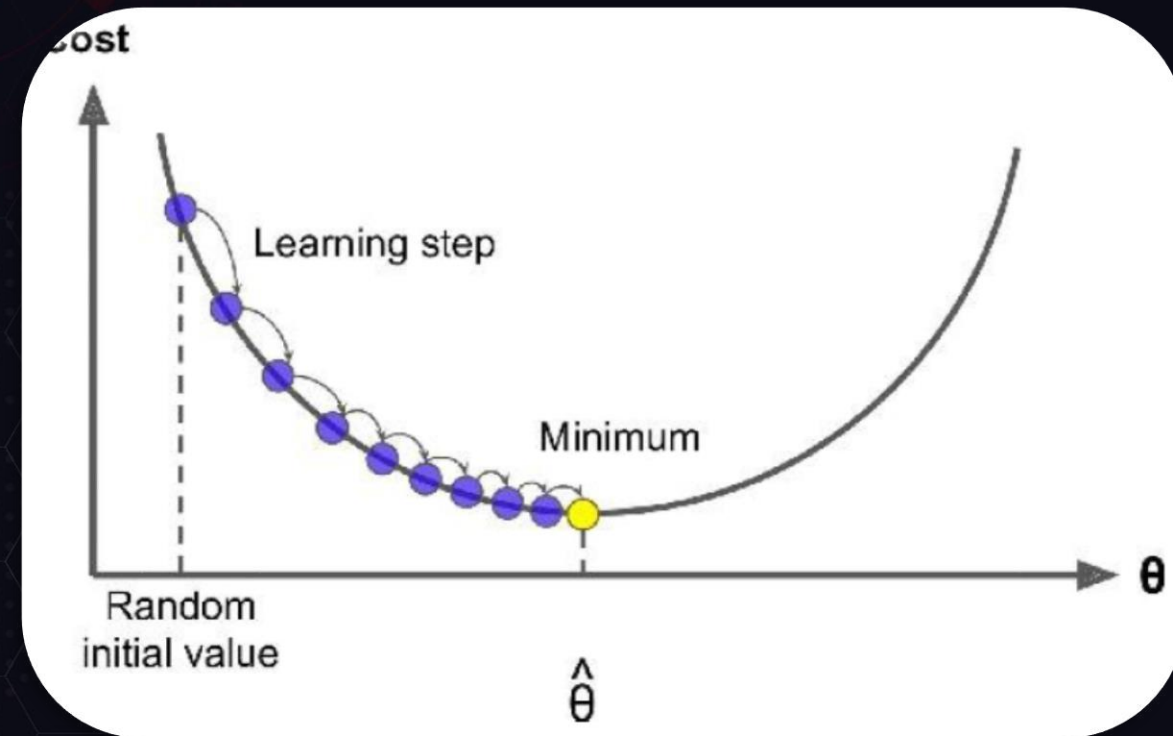
The best option for you is to start walking down the hill and trying to find the minimal distance in each of your steps towards downwards and finally reaching the bottom of the mountain.

This is actually how the Gradient Descent works. It measures the local gradient of the Error Function with regards to the parameter vector θ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum.

Gradient Descent – Cont.

Concretely, you start by filling θ with random values (this is called random initialization), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.

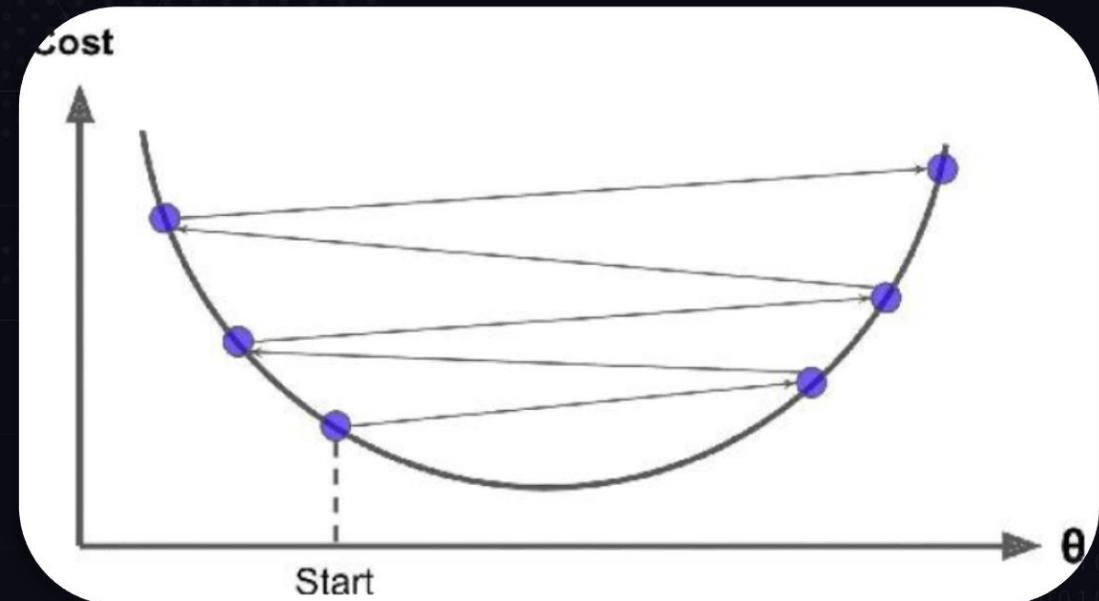
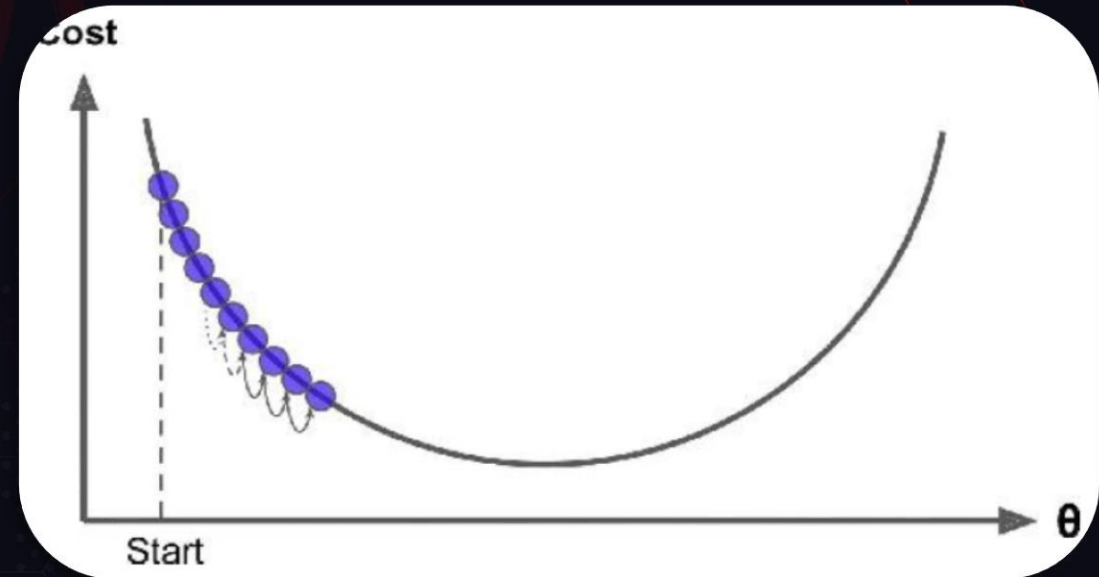
An important parameter in Gradient Descent is the size of the steps, determined by the ***Learning rate*** hyperparameter.



Gradient Descent – Learning Rate

If the Learning Rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time.

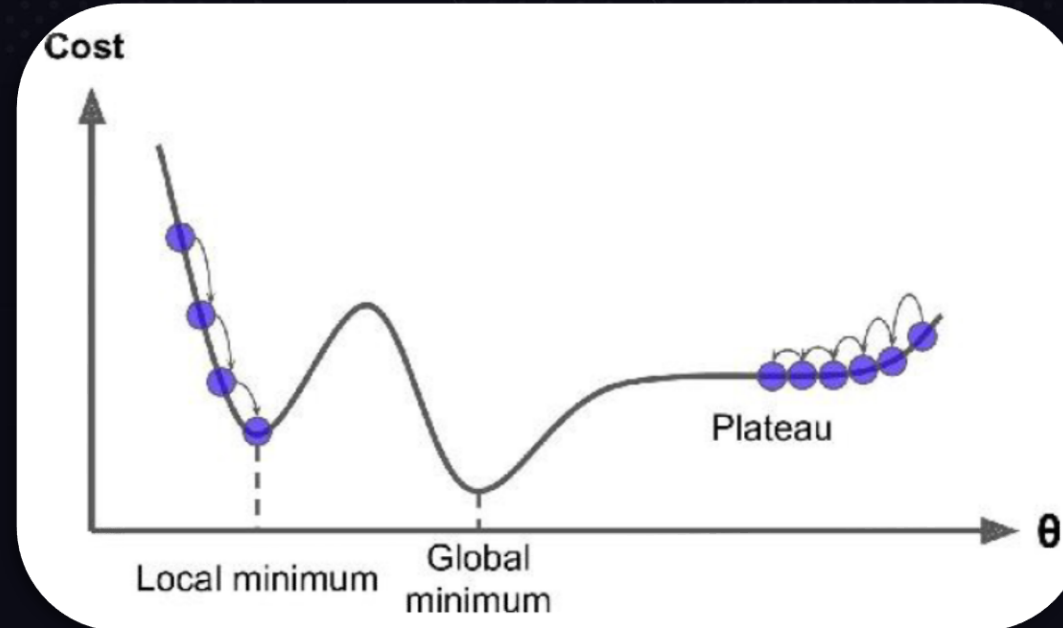
On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution



Gradient Descent – Learning Rate

Finally, not all cost functions look like nice regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum very difficult. The picture below shows the two main challenges with Gradient Descent –

- If the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global minimum.
- If it starts on the right, then it will take a very long time to cross the plateau, and if you stop too early you will never reach the global minimum.

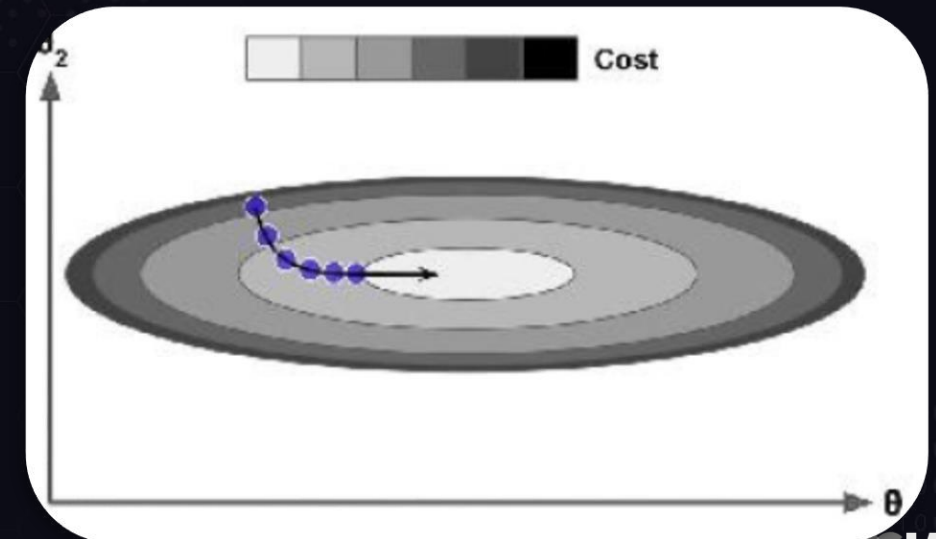
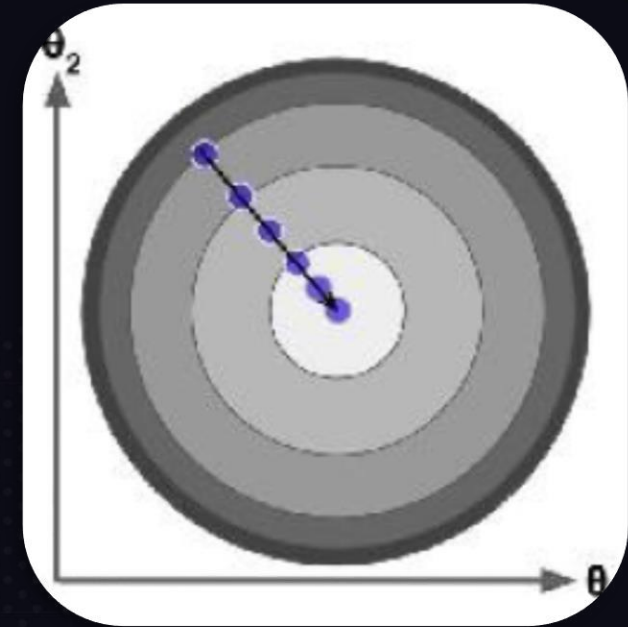


Gradient Descent & Feature Scaling

The MSE cost function for a Linear Regression model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve.

This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly. Gradient Descent is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).

In fact, the cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. The picture shows Gradient Descent on a training set where features 1 and 2 have the same scale and on a training set where feature 1 has much smaller values than feature 2.



Gradient Descent & Feature Scaling – Cont.

As you can see, on the left the Gradient Descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's parameter space: the more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in three dimensions. Fortunately, since the cost function is convex in the case of Linear Regression, the needle is simply at the bottom of the bowl.

When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

Batch Gradient Descent

To implement Gradient Descent, we need to compute the gradient of the cost function with regards to each model parameter θ_j . In other words, we need to calculate how much the cost function will change if you change θ_j just a little bit. This is called a partial derivative.

It is something like asking “what is the slope of the mountain under my feet if I face east?” and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). The below equation computes the partial derivative of the cost function with regards to parameter θ_j .

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

Batch Gradient Descent – Cont.

Instead of computing these partial derivatives individually, we can use the equation below to compute them all in one go. The gradient vector, noted $\nabla_{\theta} \text{MSE}(\theta)$, contains all the partial derivatives of the cost function (one for each model parameter)

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

Notice that this formula involves calculations over the full training set X , at each Gradient Descent step! This is why the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every step. As a result it is terribly slow on very large training sets. However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation.

Batch Gradient Descent & Learning Rate - Coding

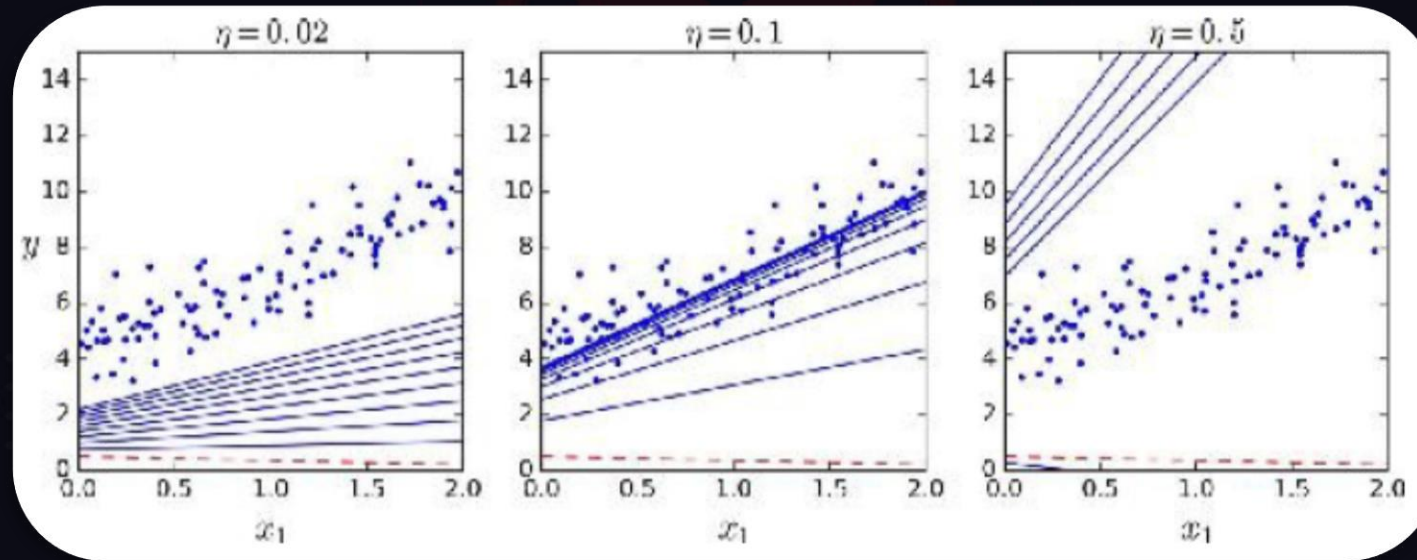
Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting $\nabla_{\theta} \text{MSE}(\theta)$ from θ . This is where the learning rate η comes into play. We have to multiply the gradient vector by η to determine the size of the downhill step.

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

```
eta = 0.1 # learning rate
n_iterations = 1000
m = 100
theta = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

Output: $\begin{bmatrix} 4.21509616 \\ 2.77011339 \end{bmatrix}$ (Nearby to $4 + 3x$)

Batch Gradient Descent – Change over Learning Rate



Here, the first 10 steps of Gradient Descent using three different learning rates (the dashed line represents the starting point).

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few iterations, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

Batch Gradient Descent – Convergence Rate

When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), it can be shown that Batch Gradient Descent with a fixed learning rate has a **Convergence Rate** of –

$O(1 / \text{Iterations})$

In other words, if you divide the tolerance ϵ by 10 (to have a more precise solution), then the algorithm will have to run about 10 times more iterations.



Topic - 03

Training Models

Part: 03



+88 0172 6867 984



tanvir@socian.ai

Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. On the contrary, Stochastic Gradient Descent (SGD) just picks a random instance in the training set at every step and computes the gradients based only on that single instance.

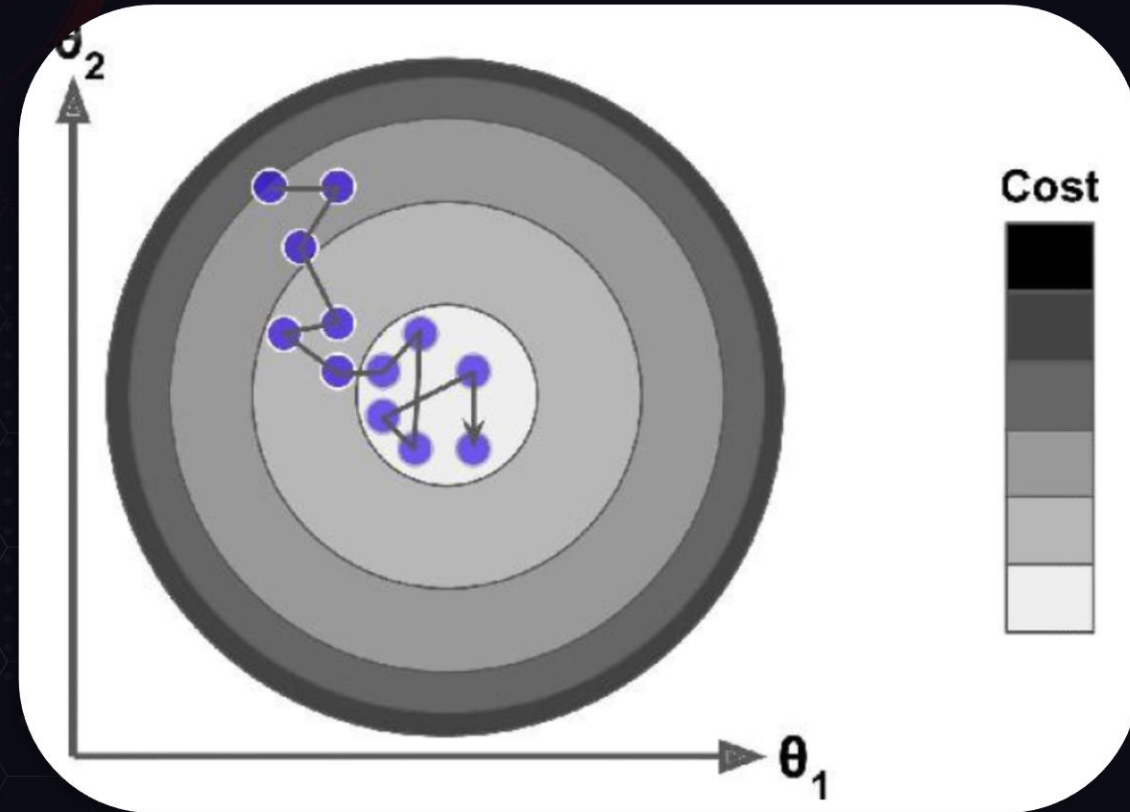
This makes the algorithm much faster since it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.

On the other hand, due to its stochastic (i.e., random) nature, SGD is much less regular than Batch Gradient Descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. So once the algorithm stops, the final parameter values are good, but not optimal

Stochastic Gradient Descent – Cont.

When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum than Batch Gradient Descent does.

Therefore randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum.



Stochastic Gradient Descent – Cont..

One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.

This process is called ***Simulated Annealing***, because it resembles the process of annealing in metallurgy where molten metal is slowly cooled down.

The function that determines the learning rate at each iteration is called the learning schedule. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum.

If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

Stochastic Gradient Descent – Coding

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1) # random initialization
for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

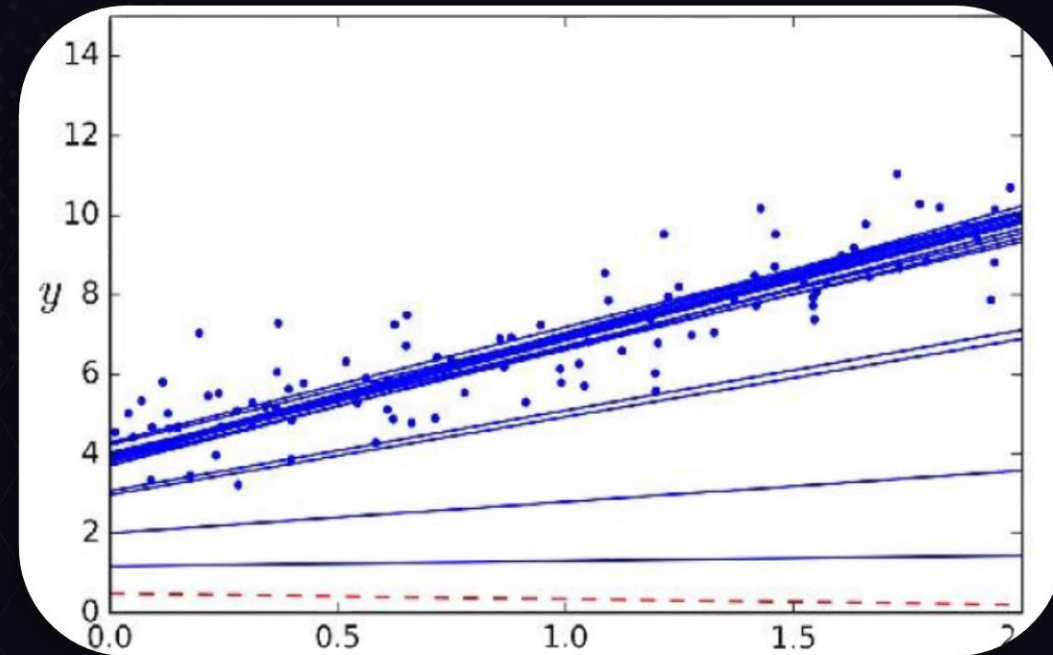
Output: `[[4.21076011], [2.74856079]]`

Stochastic Gradient Descent – Cont...

By convention we iterate by rounds of m iterations; each round is called an epoch. While the Batch Gradient Descent code iterated 1,000 times through the whole training set, this code goes through the training set only 50 times and reaches a fairly good solution.

If we look into the first 10 iterations of SGD, we can see how irregular the steps are.

Since instances are picked randomly, some instances may be picked several times per epoch while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, we can shuffle the training set, then go through it instance by instance, then shuffle it again, and so on. This is slower.



Stochastic Gradient Descent – Using Scikit Learn

If we use Scikit Learn, the equivalent code looks like this:

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
print(sgd_reg.intercept_)
print(sgd_reg.coef_)
```

The output that we get would be like the following:

```
[4.16782089]
[[2.72603052]]
```

The above code runs 50 epochs, starting with a learning rate of 0.1 (eta0=0.1), using the default learning schedule

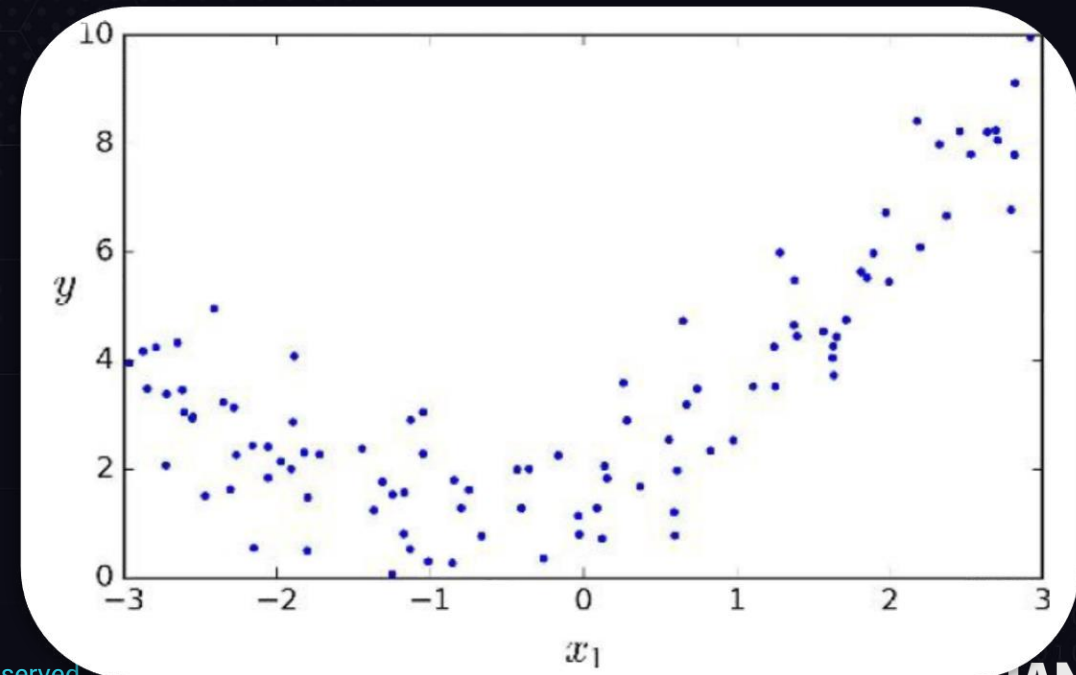
Polynomial Regression

In case of complex data, that can not be separated by a Straight Line, what can we do?

You can actually use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

Let's generate some nonlinear data, based on a simple quadratic equation –

```
m = 100  
X = 6 * np.random.rand(m, 1) - 3  
y = 0.5 * X**2 + X + 2 +  
np.random.randn(m, 1)
```



Polynomial Regression – Cont.

Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (2nd-degree polynomial) of each feature in the training set as new features (in this case - one feature)

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
```

```
X_poly = poly_features.fit_transform(X)
print(X[0]) # Output: [-0.75275929]
print(X_poly[0]) # Output: [-0.75275929, 0.56664654]
```

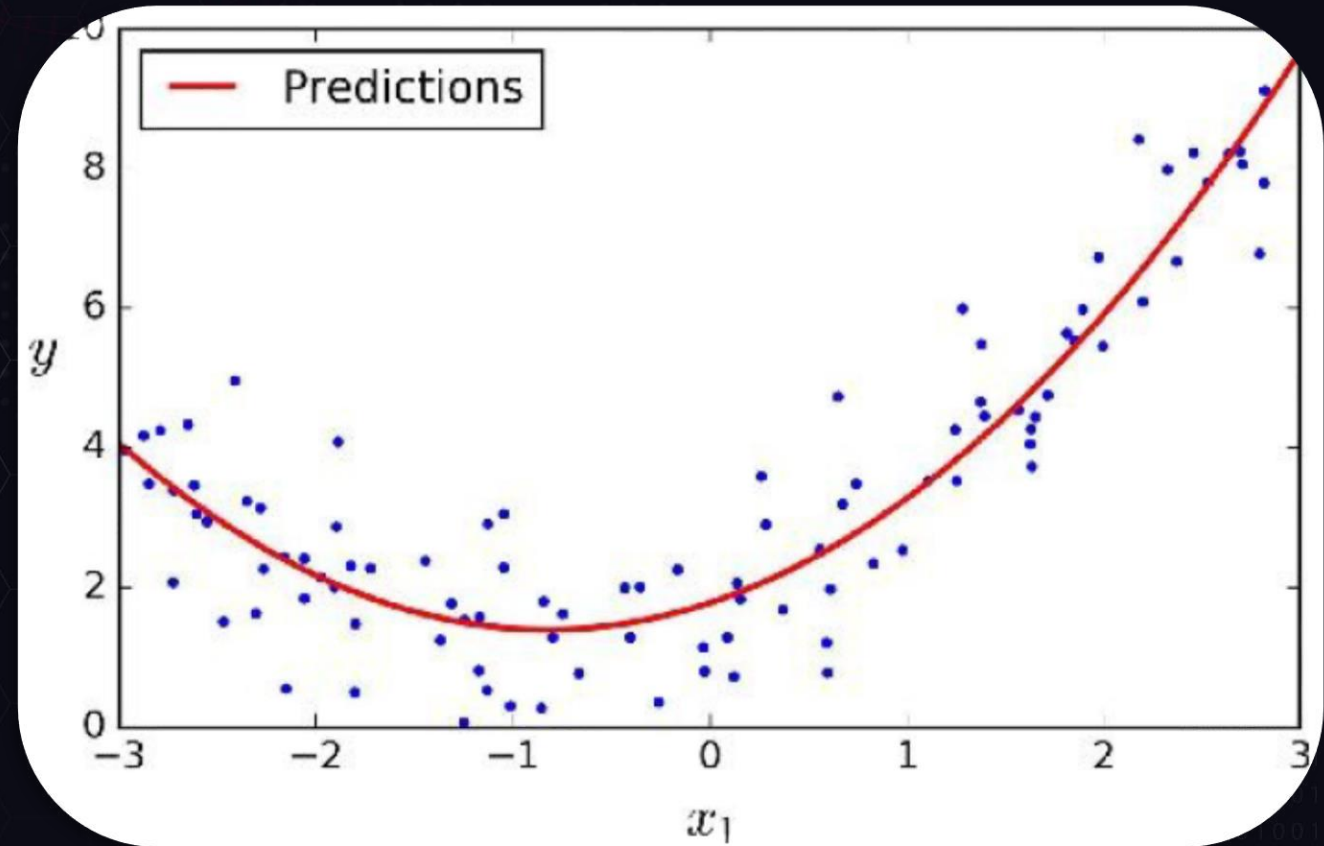
`X_poly` now contains the original feature of `X` plus the square of this feature.

Now we can fit a `LinearRegression` model to this extended training data

Polynomial Regression – Cont..

```
lin_reg = LinearRegression()  
lin_reg.fit(X_poly, y)  
print(lin_reg.intercept_)  
print(lin_reg.coef_)
```

```
[ 1.78134581]  
[[ 0.93366893, 0.56456263]]
```



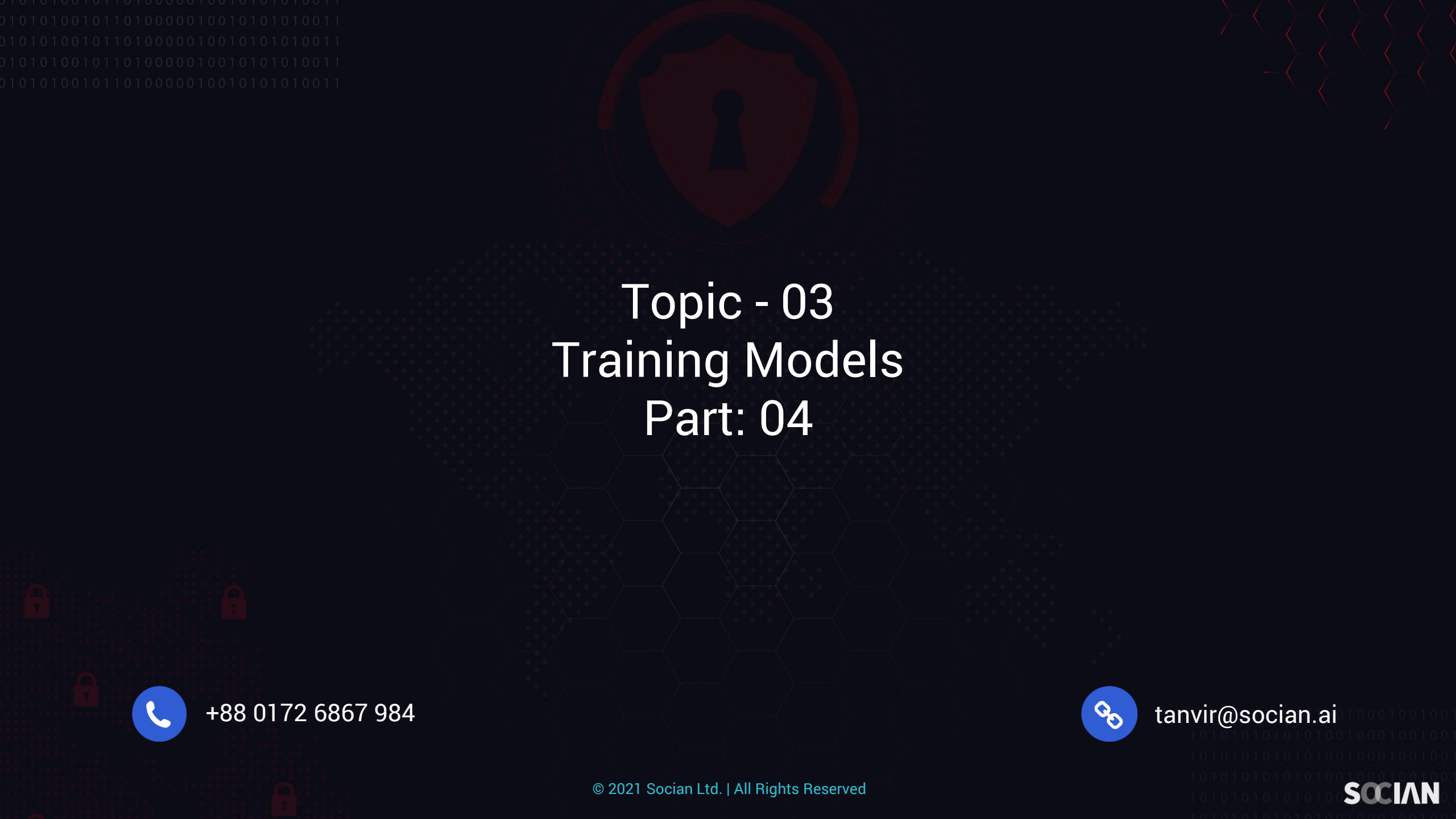
Polynomial Regression – Cont...

When there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do).

This is made possible by the fact that PolynomialFeatures also adds all combinations of features up to the given degree.

For example, if there were two features a and b , PolynomialFeatures with degree=3 would not only add the features a^2 , a^3 , b^2 , and b^3 , but also the combinations ab , a^2b , and ab^2 .

PolynomialFeatures(degree= d) transforms an array containing n features into an array containing $(n+d)!/n!d!$ features, where $n!$ is the factorial of n , equal to $1 \times 2 \times 3 \times \dots \times n$. All these features can cause the combinatorial explosion of the number of features!



Topic - 03

Training Models

Part: 04



+88 0172 6867 984



tanvir@socian.ai

Support Vector Machine (SVM)

A Support Vector Machine (SVM) is a very powerful and versatile Supervised Machine Learning model, capable of performing linear or nonlinear –

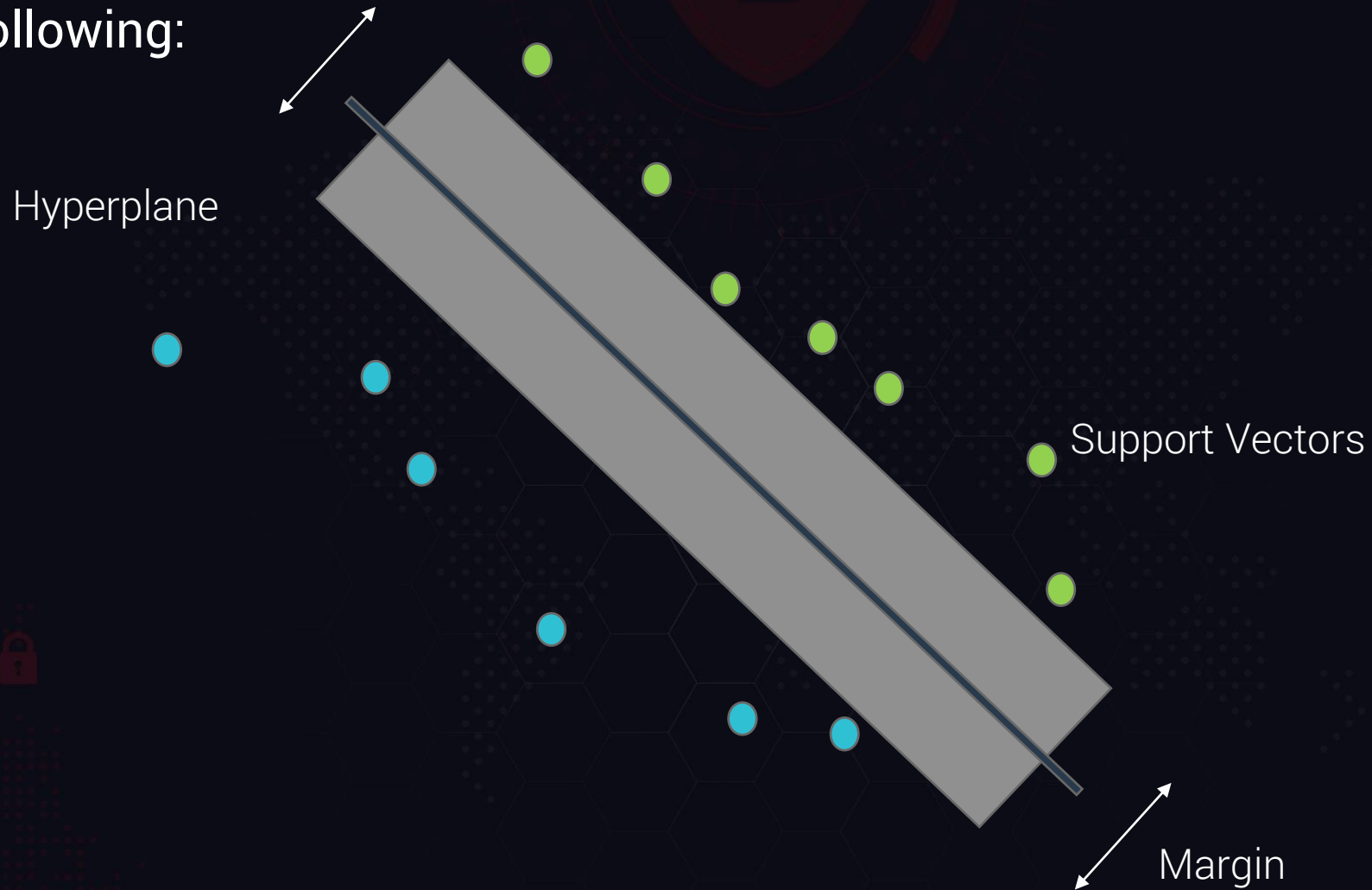
- Classification (mostly)
- Regression
- Outlier detection
- Clustering

It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have a clear idea of how SVM works.

SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.

How SVM Looks

If we look into the very basic of SVM, which is Linear SVM, then the structure looks like the following:

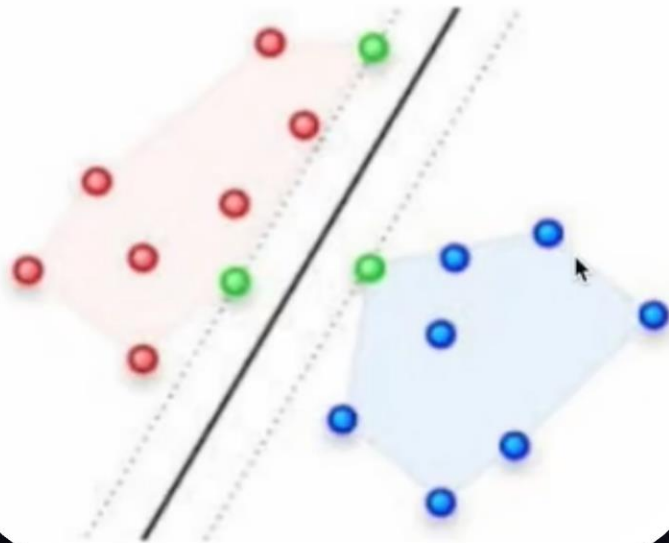


SVM vs ANN

If we look into the very basic of SVM, which is Linear SVM, then the structure looks like the following:

- SVM

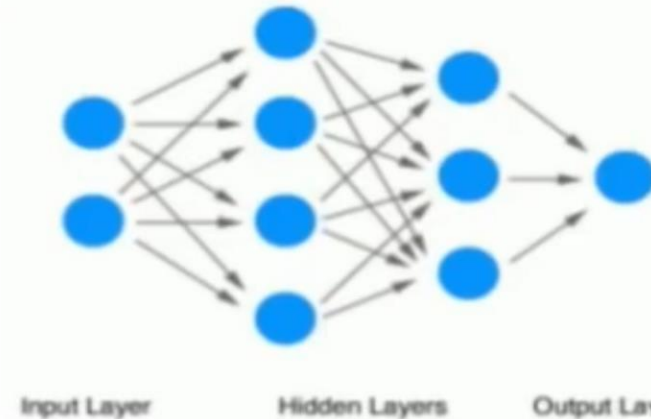
Support vector machines are relatively new form of supervised machine learning.



Works well on small data

- Artificial neural networks

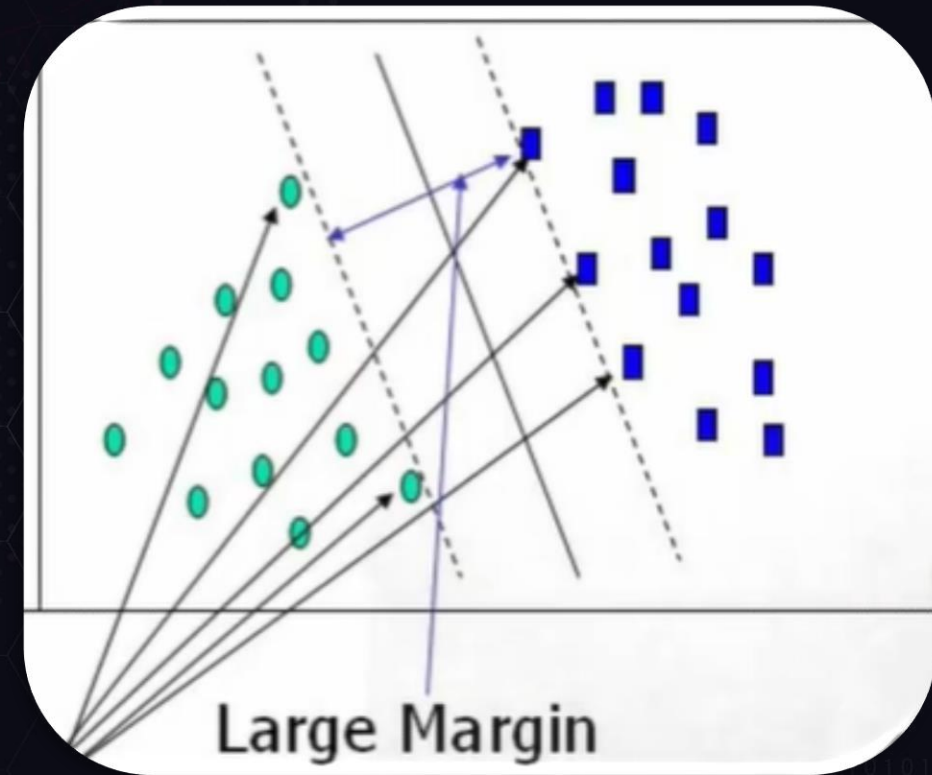
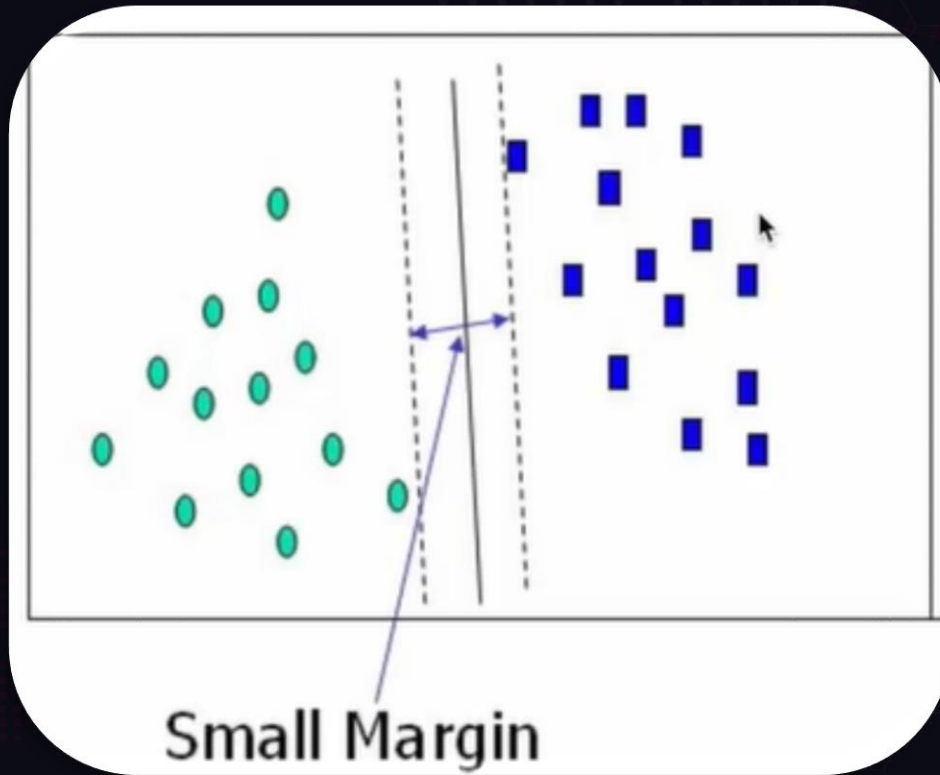
Artificial neural network by their model mimics human brain structure.



Works well on large data

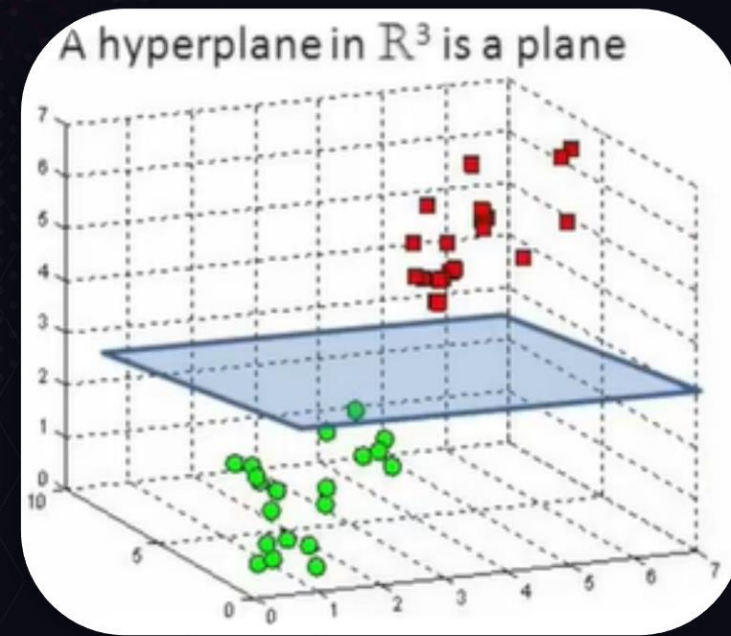
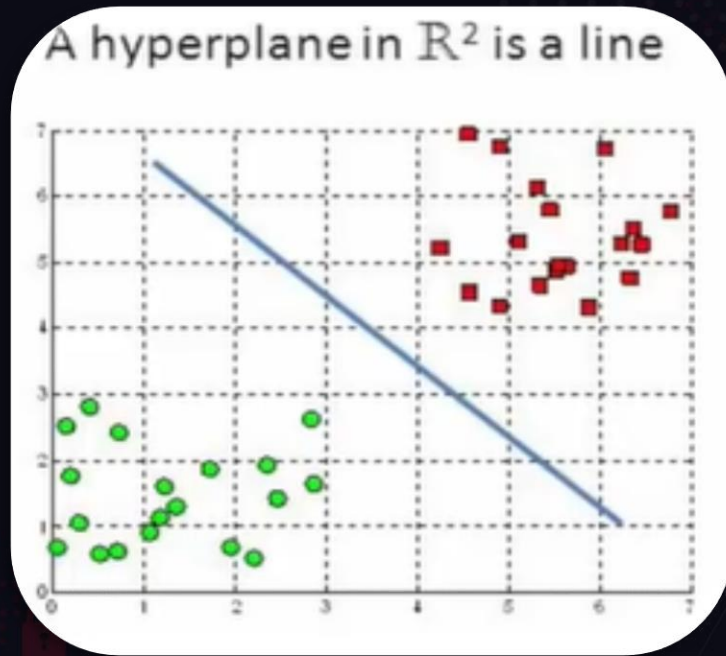
How SVM Works

If we give a SVM model two or more than two labelled classes of data, it tries to separate the data with the maximum possible width by a separator called the Hyper plane. Our goal is to maximize the Space among the Classes of Data.



What is Hyperplane

A Hyperplane is a Linear decision surface that splits the surface into 2 parts. It is quite visible that a Hyperplane is a binary classifier.



With n Dimensions of Data or n elements of feature, Hyperplane is always in $n-1$ dimensional subspace. A hyperplane of \mathbb{R}^3 features is in \mathbb{R}^2 plane.

Kernel Trick

Sometimes our Data is linearly separable and we can separate them with a straight line. Or a Multidimensional hyperplane where the data is linearly separable.

But in many cases, the data is not. Fortunately, SVM handles that for us. This is called Kernel Trick.

