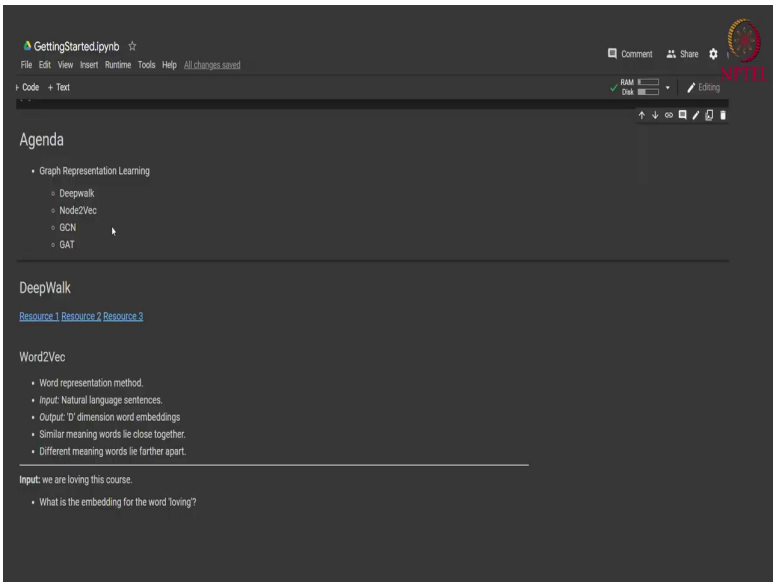
Social Network Analysis
Prof. Shivani Kumar
Department of Computer Science and Engineering
Indraprastha Institute of Information Technology, Delhi

Lecture - 05
SNA Tutorial - 05

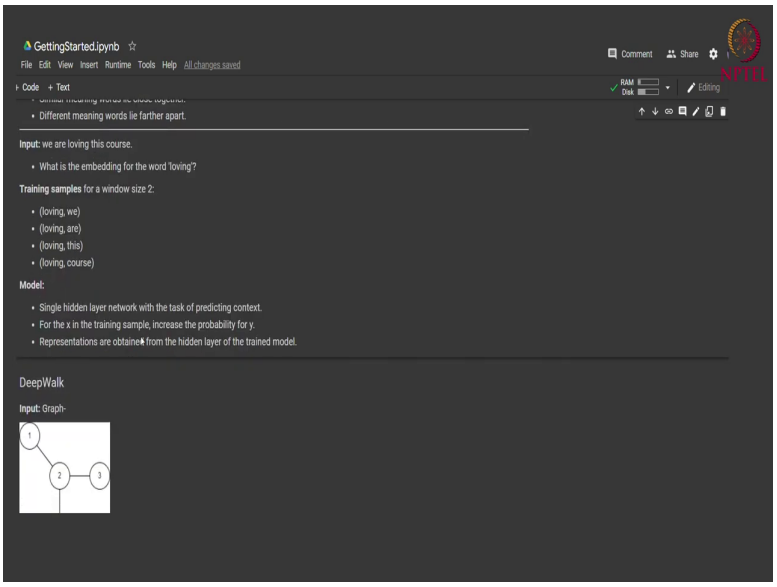
Hello everyone, welcome back to another Tutorial for their course Social Network Analysis. In today's lecture, we will study how we can basically represent graphs and nodes in a computational manner. So, we have already seen the theoretical side of it, we have learnt various algorithms like DeepWalk, Node2Vec and GCN etcetera.

(Refer Slide Time: 00:46)



In today's class, we will be learning how can we use these algorithms in a coding manner. So, we will start with DeepWalk, then we will go to Node2Vec, further we will see how we can implement GCN and graph attention networks.

(Refer Slide Time: 01:07)



So, to start with DeepWalk, we let us just quickly brush up our concepts of DeepWalk. So, to understand DeepWalk, we must know what basically is Word2Vec because DeepWalk and Node2Vec both of them follows a similar procedure, follow the similar concept as the Word2Vec mechanism. So, a Word2Vec is basically a you know a mechanism, where we can learn word representations.

It is for normal natural language sentences, it was not based on graphs. So, in the normal Word2Vec scenario, the input are basically natural language sentences and the output that we expect are the vector representation for all the words that are present in the input sentences. So, but so, for example, let us take an example. So, for example, let us consider that the input is we are loving this course and the output that we want is the embedding for the word loving.

Now, Word2Vec can be learned in you know by using two types of mechanisms; one is skip-gram and another is bag of words method. So, we will stick to skip-gram ins this in this particular class, because DeepWalk and Node2Vec they both follow this skip-gram kind of mechanism. So, in this skip-gram mechanism what we try to learn is that given a particular word, we want to learn the context of that word.

For instance, here in order to learn the embedding for the word loving, the training samples that are created for our you know for our deep learning model looks something like this. So, we have loving with the word we; so, suppose here the window size that is the number of words we want to consider in the context is 2. So, what we will do is we will consider 2

words coming after the target word and 2 words coming before the target word right; so, that makes our window size.

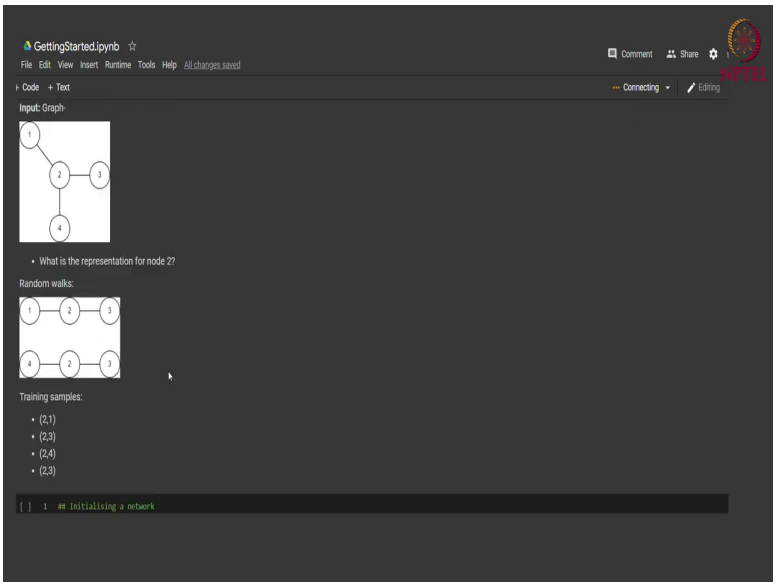
So, since the word since the word comes in between of our sentence, we have 2 words before it and 2 words after it. So, the training sample will be created in that manner only, we have loving and then the first word in its context that is we. Then we have loving that is x again like the input of the model loving and another word that is coming in its context such as are, similarly we have the other two training samples that is this and course.

Now, in the Word2Vec mechanism what will happen is we learn a simple feed forward network which will consist of one hidden layer, that hidden layer will contains for example, D number of neurons. If the hidden layer contains D neurons, we will get a vector representation of size D. So, we will see how it happens. So, basically we have this three layer feed forward network; the input, the hidden layer and the output layer.

And the aim, the task that we train this feed forward network is basically to predict the probability of a particular word to occur in the vicinity in the required window size of the input word. So, for example, here for the input word loving we want the probabilities for the word we are this and course to be higher than the probability of all the words that are present in the vocabulary. So, using this problem statement and this task, we train this 3 layer feed forward network.

And, after we have trained this model we just take the weights of the hidden layer to be our vector representation for a particular word right. So, this is basically how Word2Vec works. Now DeepWalk, it follows a similar strategy, but in the; so, in Word2Vec, the input the inputs were basically words, were sentences, but in DeepWalk the input is basically a graph.

(Refer Slide Time: 05:54)



Now, in order to represent a graph in a way that the words were represented for Word2Vec, what we what the authors of this paper, this method suggested is the is that we use something called random walks. So, you are already aware what are random walks. So, for instance I have taken this very simple example here, suppose this is the graph that we are considering and we want to find out the representation for the node 2.

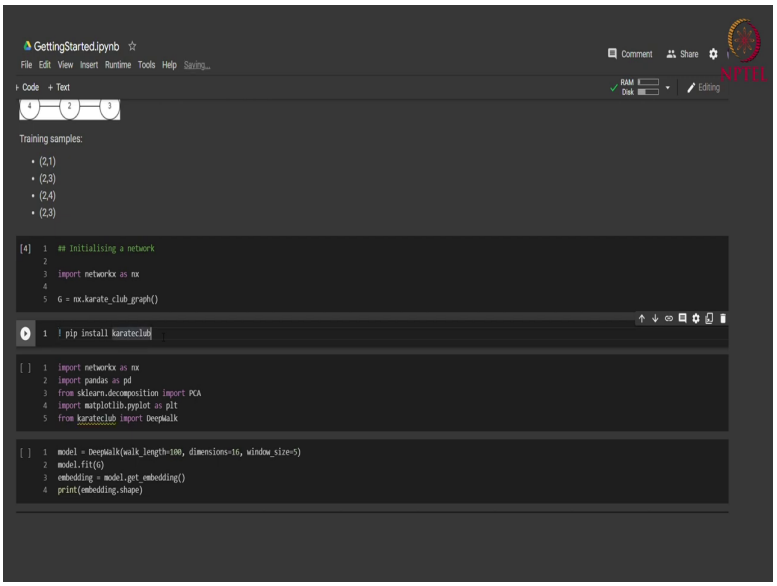
So, in this case what we will do is we will create these random walks for this particular graph. So, we have these two random walks, where 2 is coming in between of the walks. And, similar to Word2Vec we create these training samples such that the x part of the training samples the input is 2, that is a target node for whichever the representation we want and the y part are basically the neighboring nodes based on the window size that we want.

So, suppose here the window size is 1; so, we have the training samples like this that for the node 2 we have 1 in our vicinity, for the node 2 again we have 3, then for 2 we have 4 and 3 again based on these random walks. Then, we use these training samples to learn a skip-gram kind of strategy, in order to learn the representation for all the nodes.

So, now we can see what is happening is basically that the similar meaning words or similar meaning nodes or you know like the node which capture a similar concept or lies in you know in each others neighborhood, those will have a representation that is quite similar to each other. Whereas, different meaning words or in the case of graphs, the nodes which lie farther apart they will have you know they will have very different representations.

So, that you know now how can we decide which representation is similar or different? So, we have various different distance measures like cosine similarity or other distance measures right. So, based on those distance measures, if we learn a DeepWalk DeepWalk you know representation using this skip-gram strategy, we can say that similar neighborhood nodes will have similar representation, whereas, farther apart knows will have different representation.

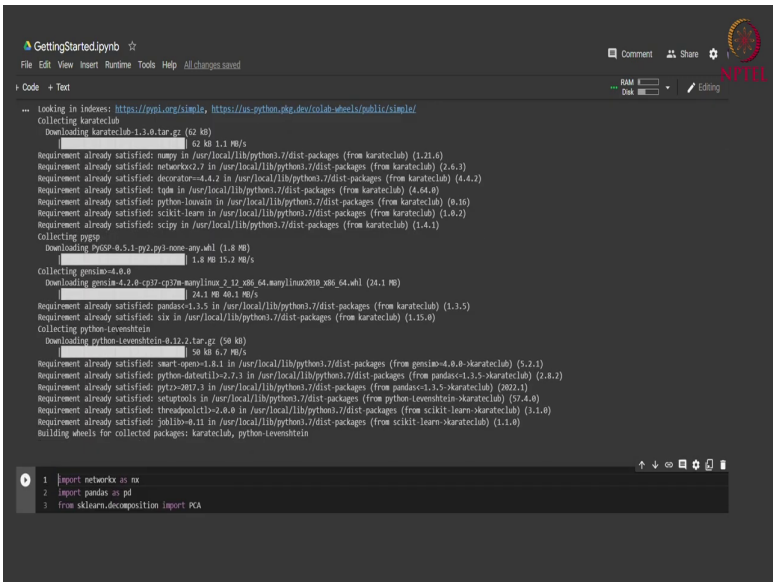
(Refer Slide Time: 08:22)



Now, how to do that? How to implement it in Python. So, what we will do is we will just initialize a sample graph. We already we already know about this karate club graph, we have used this graph in our previous tutorials. So, we will use the same graph here and we will just initialize it in a graph object G.

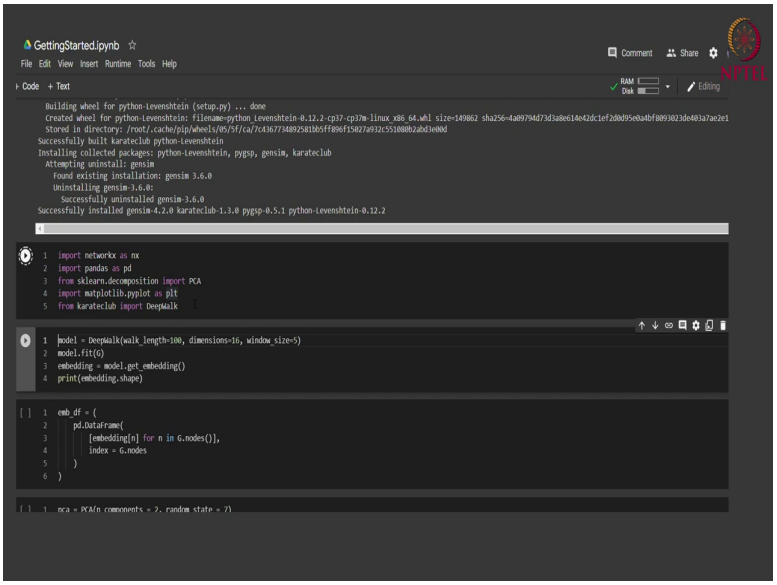
Then, what we will do is we will use this karate club library which is basically you know a helper library for network x which contains various machine learning algorithms in it. So, this library also contains a method for you know using which we can get the DeepWalk representation. So, in order to use this library, we will first need to install this library. So, we will just do a quick pip install.

(Refer Slide Time: 09:10)



And, once we you know once we install this library, then we will be using this library.

(Refer Slide Time: 09:17)

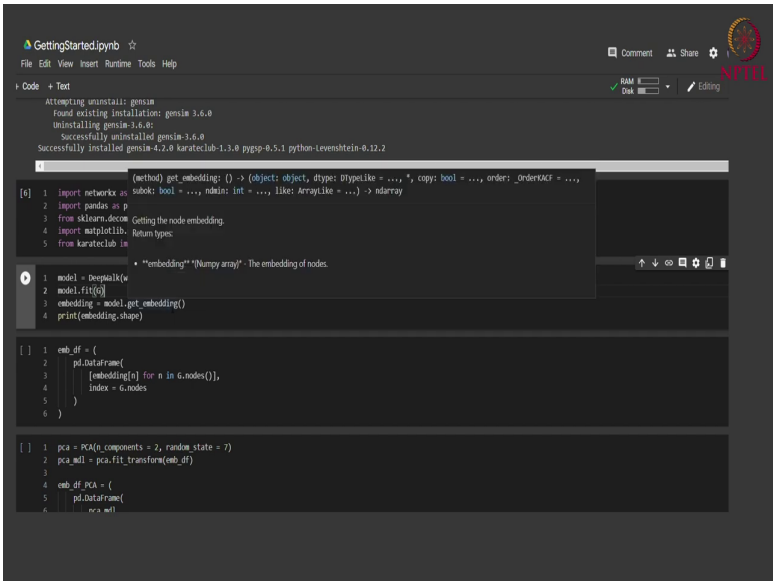


So, yeah so, this library is installed. And, now how can we use this library? We will just simply import it. And, since we want to use the DeepWalk mechanism of this library, the DeepWalk algorithm from this library, we will just import the DeepWalk method from the karate club library. Then, we import some other helper libraries that we need.

(Refer Slide Time: 09:44)

So, we import that and now how can we use this DeepWalk function basically is that first we need to initialize a deep DeepWalk model. And, to do that we will just call this DeepWalk method that we have imported from karate club and we pass some parameters to it.

(Refer Slide Time: 10:06)



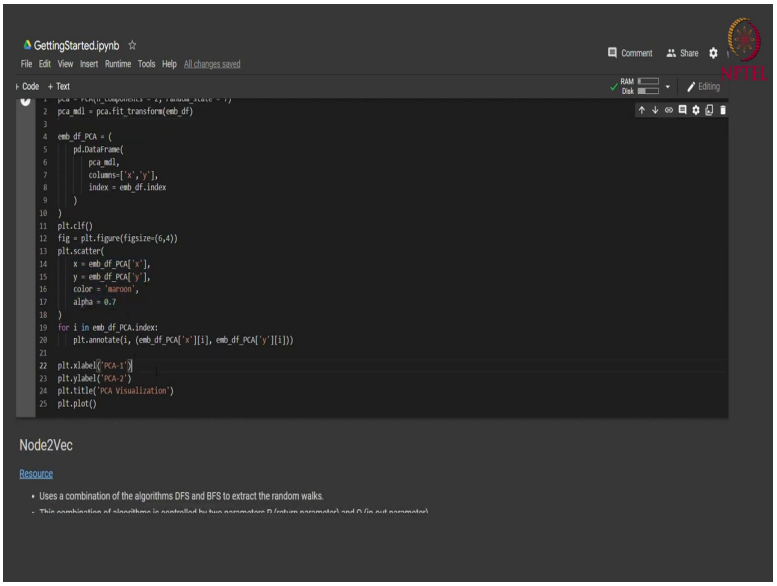
Now, these parameters they basically you know decide the window size that we want or the dimensions of the resultant embedding that we want and so on. So, and also the you know the maximum length of the random walks that we want. So, we initialize this DeepWalk method

using these parameters and we here for our you know for our use, we have just said walk length to 100, dimension to 16 and window size to 5.

So, here for all the 34 nodes that are present in the karate club graph, we will get you know a vector representation of size 16 for each of the nodes right. So, after we have initialized this model, we needed to train this model. So, we just call the model dot fit function on the graph object that we have, that is the karate club graph that we have initialized. Now, this to obtain the embeddings from this trained model, we will just call the get embedding function over this model object.

So, we have this model dot get embedding and we will be returned with this embedding object. Now, this embedding object, see you can see here each of the node, now each of this object in the embedding in the embedding variable, it is basically a vector of the dimension 16 correct; because that is the dimension that we have specified for the DeepWalk algorithm.

(Refer Slide Time: 11:37)

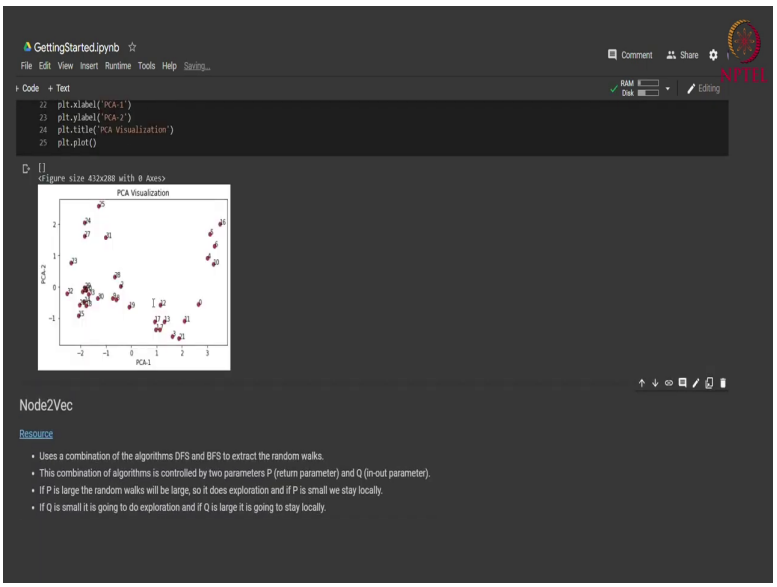


Now, we will just we will just store these embeddings in a data frame; so, that it is easier to access for the further steps that we want to perform. So, we just store these embeddings in this emdf here that we have mentioned. And, in order to visualize this embedding, now we see that each node is represent by a 16 dimension vector. Now, to visualize a 16 dimension vector is; obviously, a very difficult task and it is impossible to you know to interpret it.

So, what we do is we reduce the dimension of these vectors. So, from 16 dimension, we reduce it to a 2 dimension you know vector by using something known as principal component analysis. So, it basically finds out the principal components from the you know from the big vector and based on the resultant dimension that we want, it identifies the principal you know the top most principal components for that number.

And, then we just map the you know higher dimension vector into a lower dimension using those principal components. So, here we did that only. We initialize the PCA object with you know resultant dimension 2, because we want to visualize our vectors in a 2D space. And, we then you know transformed all our vectors of dimension 16 into a 2 dimensional space. Then, we simply just plot this plot these 2 dimensions that we have now.

(Refer Slide Time: 13:20)

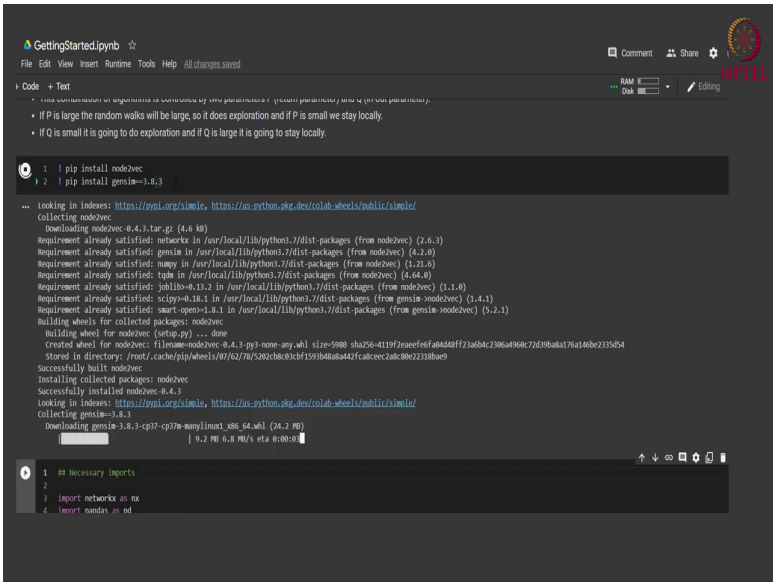


And, after we run this we can see that this is how our visualization looks. So, if you can see here, you can see that the nodes present here like 32, 30, 29, 15, they basically lie closer together right; whereas, these nodes that is 16, 5, 6, 4, 10, they lie farther from the 32, 30, 29 nodes, but closer to each other right. So, we can see that here we can see that there are four kind of clusters that are forming right.

So, this kind of visualization helps us to basically see some kind of structure that is present in the graph, that might not be you know might not be seen in the normal; if we just you know normally draw the graph using the network extra function for instance. Now, here we are you know the result is that we have is a 16 dimension embedding for each of the node using the

DeepWalk mechanism. Now, these embeddings can be used for any you know any end tasks such as node classification or such that tasks. Now, apart from DeepWalk, we have another method known as Node2Vec right.

(Refer Slide Time: 14:40)



So, in this Node2Vec mechanism, it is extremely similar to DeepWalk, but in this mechanism the random walks that are created from the input graph, they are basically you know controlled by two parameters P and Q. Now, in the Node2Vec mechanism what happens is that the algorithm, the whole algorithm is basically a combination of DFS that is Depth First Search and BFS that is Breadth First Search and these two searches they are regulated by the parameters P and Q.

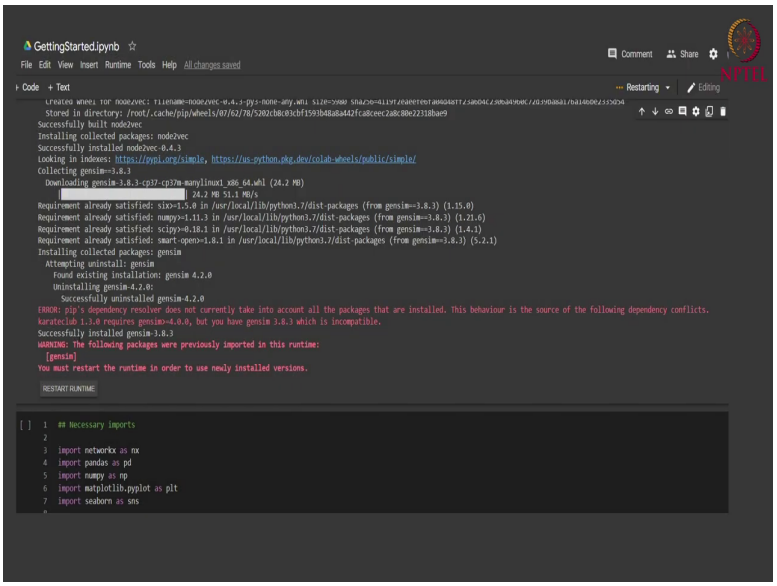
So, basically you know increasing or decreasing these parameters, it specifies whether we want a representation to be restricted locally that is the resultant representation should you know should be able to identify the local structure of the graph or based on these parameters P and Q, do we want the representation to capture the global structure or global structure of the graph or for a particular node it should capture the effect of the node that is also like that lies farther in the graph right. So, that is the high level intuition of it. We have already seen it in a deeper manner in the theory class.

So, based on these P and Q parameters, the representations, the random walks are created. And, then once we have the random walks, we use a similar mechanism like we did in DeepWalk. We train a simple skip-gram model to learn the final representation, the context

rich final representation. So, in order to implement Node2Vec in the Python format, we need a library which has this function Node2Vec. So, we just install this Node2Vec library.

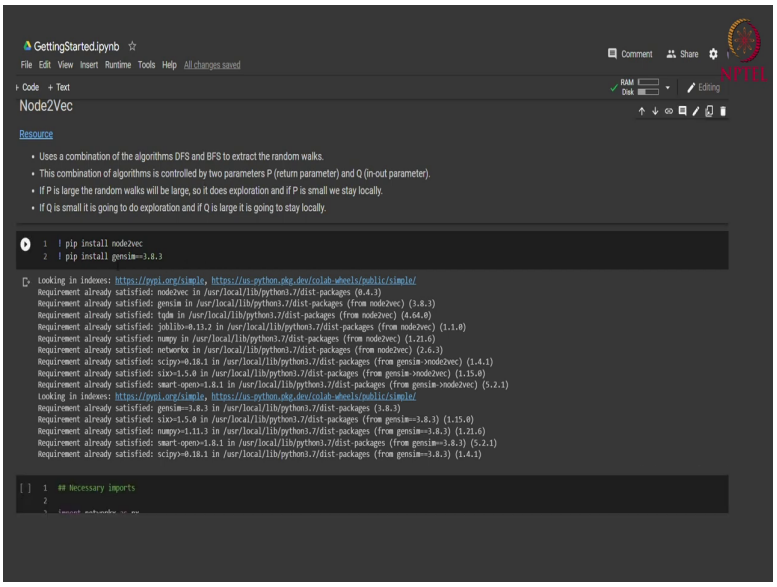
And, we just you know regulate a gensim version here so, that it is compatible with the Node2Vec library that we want right. So, here its just initializing, you might need to restart your runtime here. It might happen that we need to restart our runtime here, if you are working in colab and you are installing gensim like this.

(Refer Slide Time: 16:58)



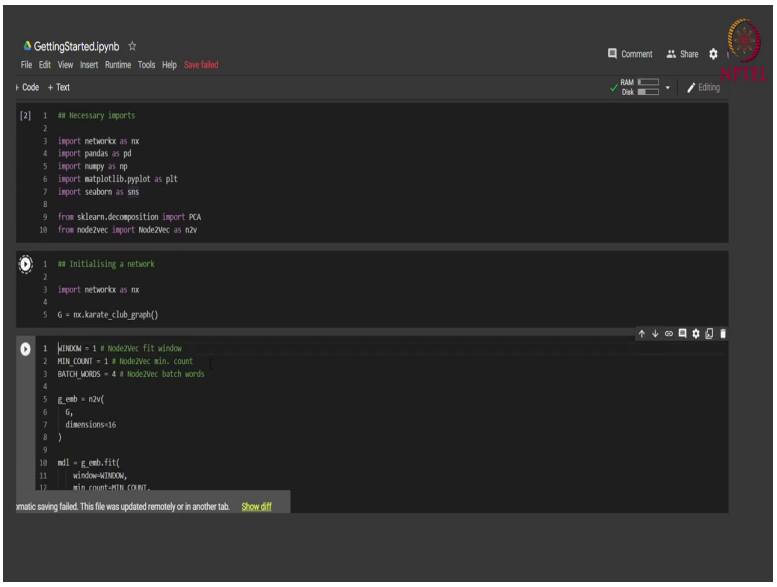
See here it is asking us to restart the runtime. So, we will be restarting the runtime here and then again if you once you have just you know here you will be able to see that the runtime has restarted.

(Refer Slide Time: 17:12)



If you will run in this, it would already show that requirement is already satisfied because we have you know we have already installed these things.

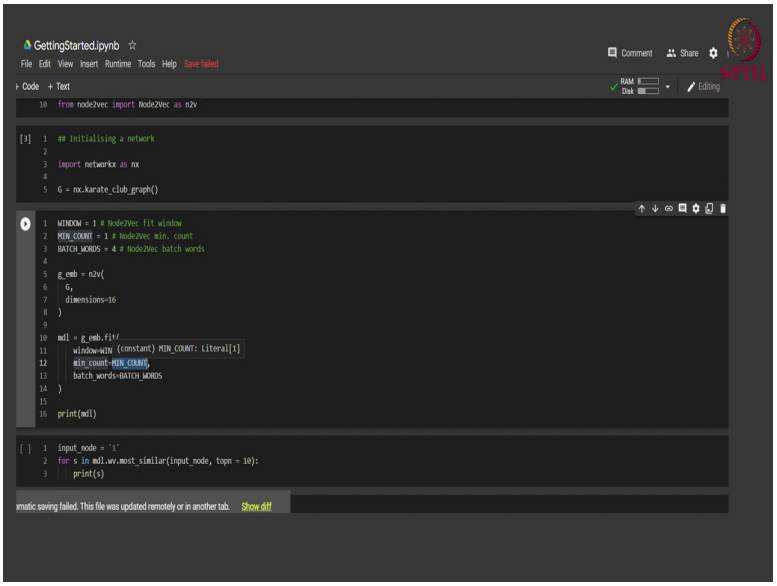
(Refer Slide Time: 17:24)



Now, once we are done with this, we will do the necessary imports that we want in order to learn the Node2Vec embeddings. From this Node2Vec library, that we just installed we will we will import the Node2Vec function as n2v here. So, we just import this and other helper libraries that we want and we will see that yeah all of these are imported and once it will run it will import.

And, then just like we did in DeepWalk mechanism, we will initialize a sample graph here. So, here also we will take the karate club graph that we have already considered in our various examples. We will just initialize it using the network function x that provides.

(Refer Slide Time: 18:12)



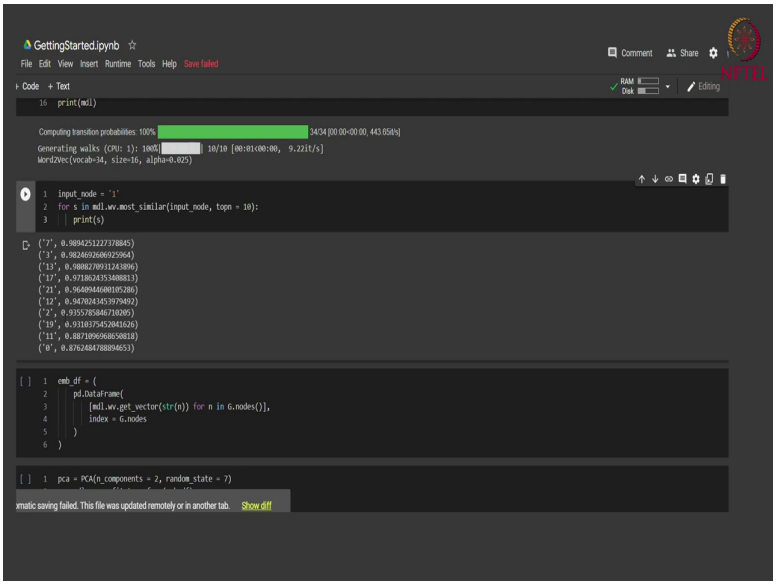
And, to learn the Node2Vec Node2Vec embeddings, we will use this n2v function that we just imported from the Node2Vec library. We will initialize this object using the input graph that we want and the required dimension of embedding that we want. So, here again we are learning a 16 dimension embedding for each of the nodes that are present in the graph. So, we initialize this object and then we learn the embeddings using the dot fit function of this object.

Then this dot fit, it basically takes in some parameters in it. So, for example, the window size. So, here we have just passed the window to be 1, that is it will just consider the first hop neighbor of in each direction for the node and then the minimum count basically it like if you look at the Word2Vec manner. So, it is essentially learning a skip gram model right after the random walks.

So, if you look at that skip gram model; so, a minimum count it basically represents the you know the least number of times a word should occur in our inputs as to be; so, that it is considered in the vocabulary. So, in the case of graphs, in the case of Node2Vec this minimum count, it basically represents the minimum number of times a node should occur in a training sample to be considered in the vocabulary right. And, then we have the

BATCH_WORDS which basically tells us the number of batches that we should provide to each of the worker in a in our this Node2Vec mechanism.

(Refer Slide Time: 20:20)



So, we learn this model and we print this model and you can see here that we it is a you know it is fitting here. And, we can see that this is the kind of model that is being learned, that is it is essentially a Word2Vec model with vocab size of 34 because we have 34 nodes. And, for each of the nodes we have the dimension 16 and this is like a regulatory parameter.

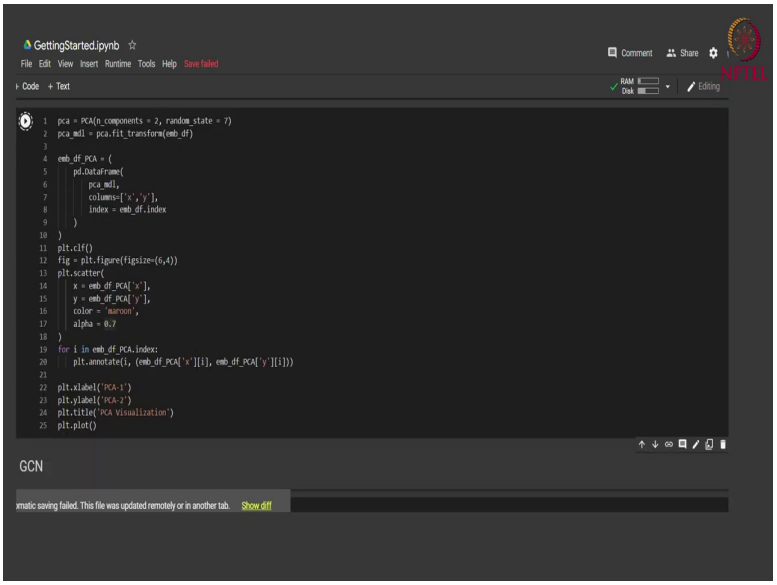
So, now, the model that is resulted by this Node2Vec library, this Node2Vec function; it works, it behaves exactly like how you would how a normal keyed vector that is learned you know that the Word2Vec. that is learned by the Word2Vec mechanism in the gensim library. This particular model that is learned by using this Node2Vec library, it behaves in the exact same manner.

So, in order you know since it behaves in the exact same manner, we can use functions like most similar in order to get the node that is most similar to the input nodes. So, for example, here we are providing the input node as 1. So, in this particular statement, we want to see the top 10 most similar nodes to the node 1.

So, we will just print this and we will see that the top 10 most similar nodes are 7, 3, 13, 17 with the similarity this. This is the cosine similarity. So, the node 1st and 7th are basically

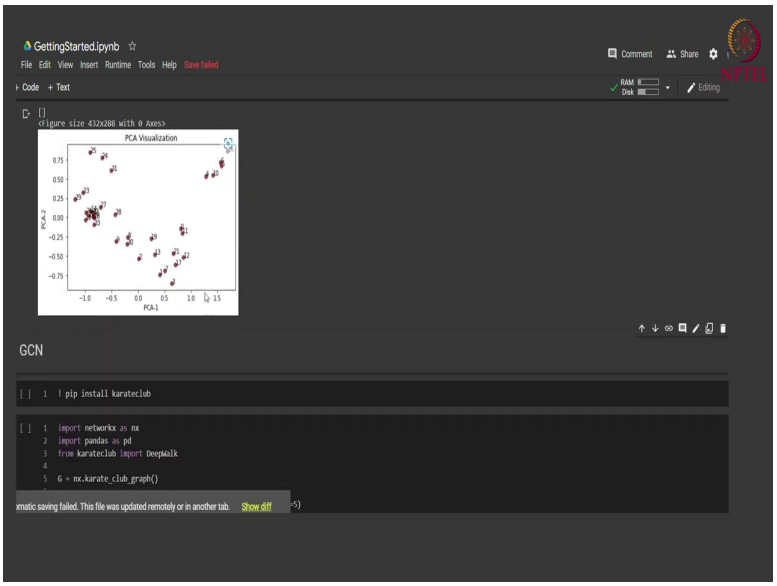
you know similar to you know up to 98.94 percent extent right and similarly we have these top 10 most similar nodes.

(Refer Slide Time: 21:52)



Now, again we in order to do a you know a neat PCA representation, we just create this data frame and then again use the same code that code snippet that we had above to create this PCA plot of the resultant embedding that we are getting from the Node2Vec mechanism ok.

(Refer Slide Time: 22:24)

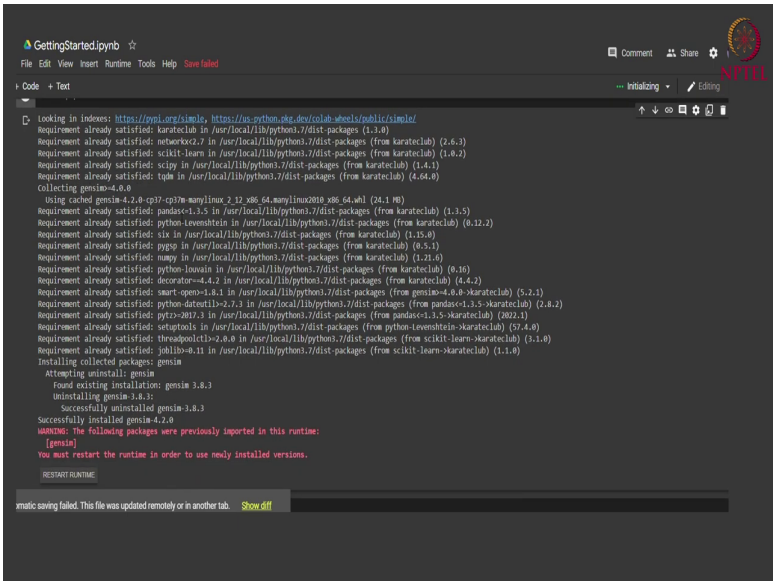


So, this particular cell is running and then what we will do is we will just quickly compare, you know this the resultant representation that we get from this PCA to the PCA that we got from the DeepWalk. So, here you can see that the nodes here 4, 10, 6, 5 and 16 like closer together 25, 24, 31 like closer together. And, if you just see the you know the PCA, that we got from DeepWalk; here we are getting like a similar kind of cluster. So, like the nodes 4, 5, 6 and 16, 10 they lie closer together as we saw in this Node2Vec representation as well.

So, we can say that a similar kind of representation is being learned by DeepWalk and Node2Vec. Now, essentially both of these methods they are learning a normal feed forward feed forward network, like they are learning the representation using a feed forward mechanism. You know as we have seen in the theoretical deep learning class, we have seen that in order to capture context you know in a in an efficient manner, in a more appropriate manner; we can use you know instead of normal feed forward thing, we can use something known as convolutions.

Like we have seen convolution networks that are used in the case of images and text in order to learn the context, the local context more efficiently. So, we naturally we want to do something similar for the cases of graph. We want to learn the you know the node representation in an even more efficient manner. So, what we do is we use something known as Graph Convolution Networks or GCN in this case.

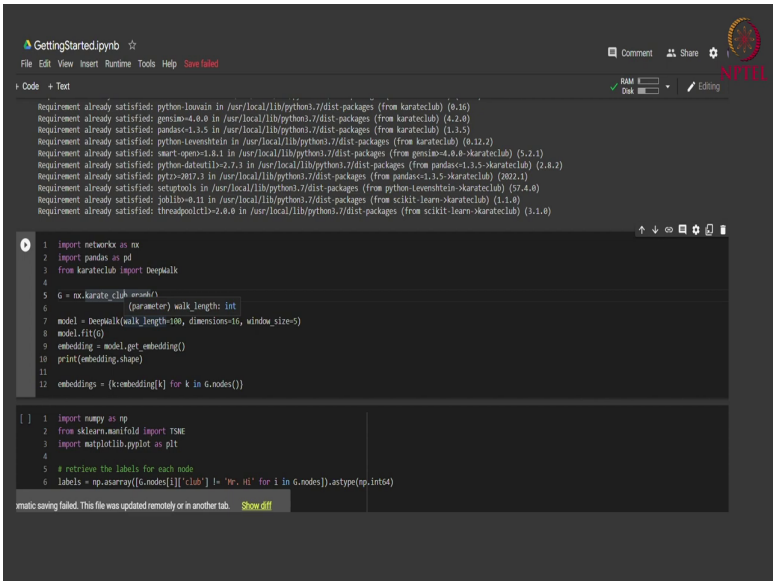
(Refer Slide Time: 24:12)



So, GCN basically it is a neural network which uses convolution kind of a thing in order to learn the representation right. And, we have already seen the theoretical aspect of GCN in the class. So, in this lecture, we will see how we can implement this GCN using PyTorch ok. So, again so, now, GCN since it is a semi-supervised kind of mechanism it needs an end task. So, the end task that we will consider in our lecture today is of node classification.

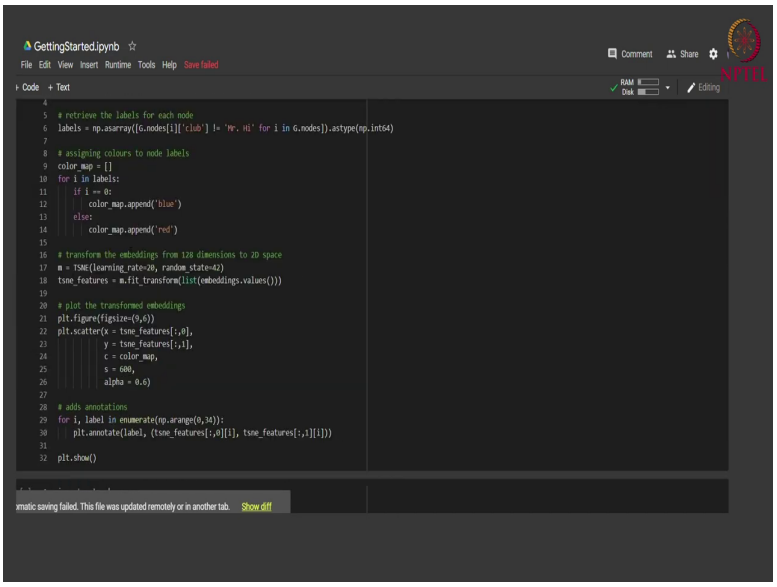
So, you already know that in this karate club graph that we are considering, each node belongs to one of the two clubs right. So, we have two clubs in the whole the graph, mister high and officer and each of the nodes belong to one of these two clubs. So, the task that we will focus on in today's lecture is to classify each node into one of these two clubs right.

(Refer Slide Time: 25:21)



So, again we just initialized this network x graph; this karate club graph from the network x library.

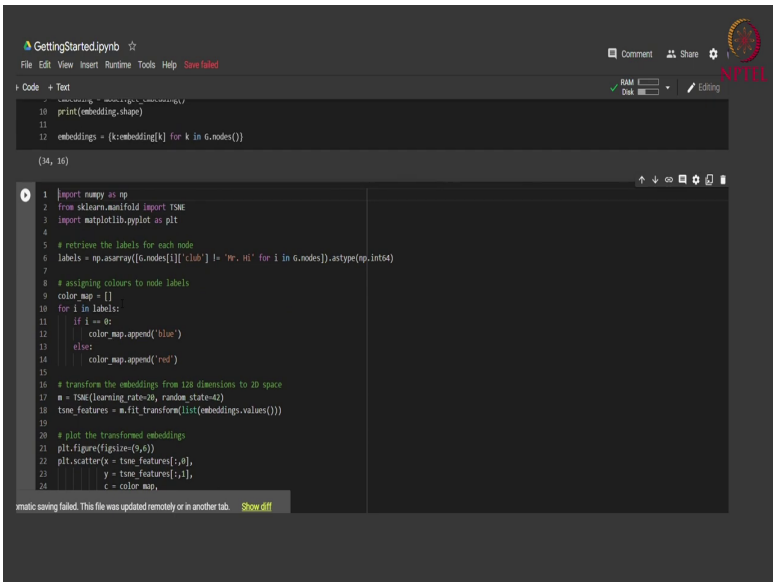
(Refer Slide Time: 25:23)



And, then we will need to again restart the runtime wait, because this gensim is now restored to the correct version ok, yeah alright. So, now, yeah so, we load this karate club graph in this G graph object. And, in order to get started with GCN, in order to make our model you know learn the node classification, we must start with some kind of initial representation for each node right. Because, in a deep learning model or any computational model, the input is are numbers only right. And, for the node we you know for in order to convert these node to numbers, we need a graph representation kind of technique.

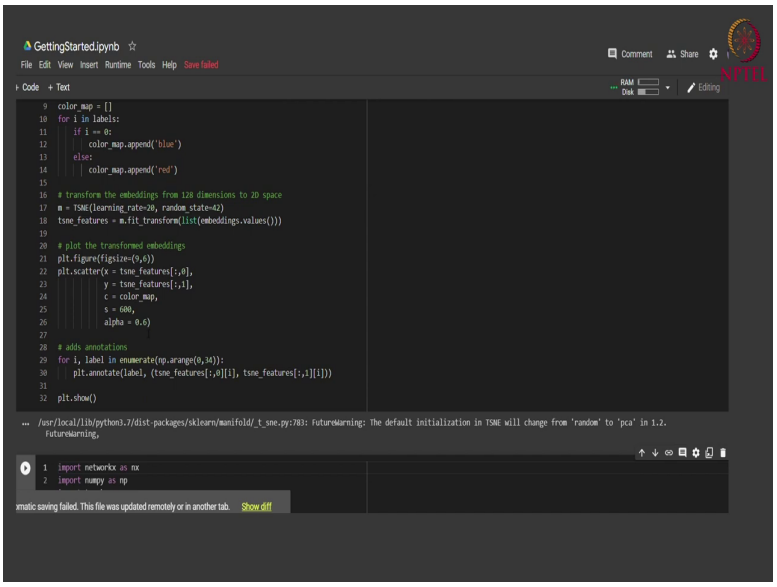
So, we might we want to initialize each of these nodes with some existing representation. So, we can use normal tfidf or normal adjacency matrix representation. But, here since we already know how DeepWalk and Node2Vec works, we are initializing our nodes with a DeepWalk embeddings right. So, this is something that we have already seen that is the DeepWalk model, that is fitted on this graph and we are just obtaining the embeddings from the learned DeepWalk model ok.

(Refer Slide Time: 26:53)



So, we are getting this embedding and the embedding shape is again 34 cross 16, that is we have 34 nodes where each node is of the dimension 16.

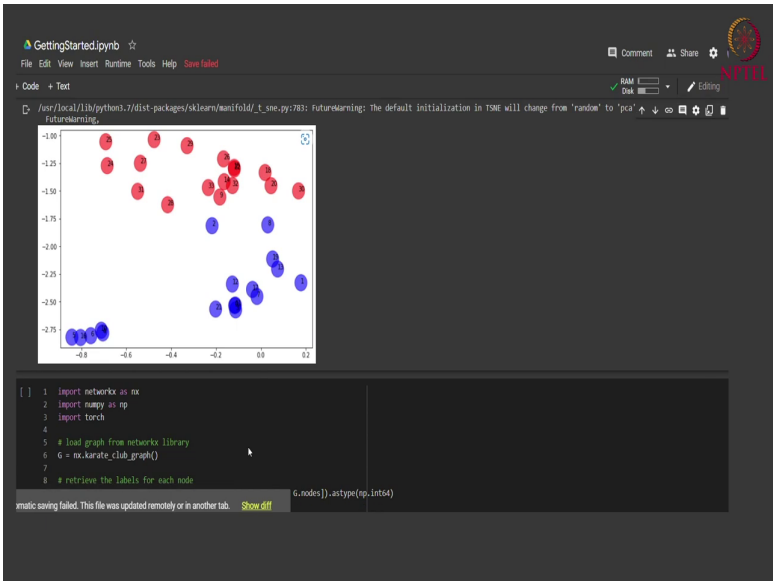
(Refer Slide Time: 27:04)



Now, what we will do is we will in order to learn this node classification method, we need to create a neural network which basically consists of GCN layers, that is graph convolution layers right. Now, before we do that, we must create a data set object for the nodes for the whole graph right, for all the nodes of the graph.

Now, to create this data set object, what we will do is ok. So, first before creating the data set, let us first just visualize the DeepWalk embeddings that we have right. So, here we are instead of PCA, we are using the TSNE plot. So, it is another dimension dimensionality reduction mechanism that we have.

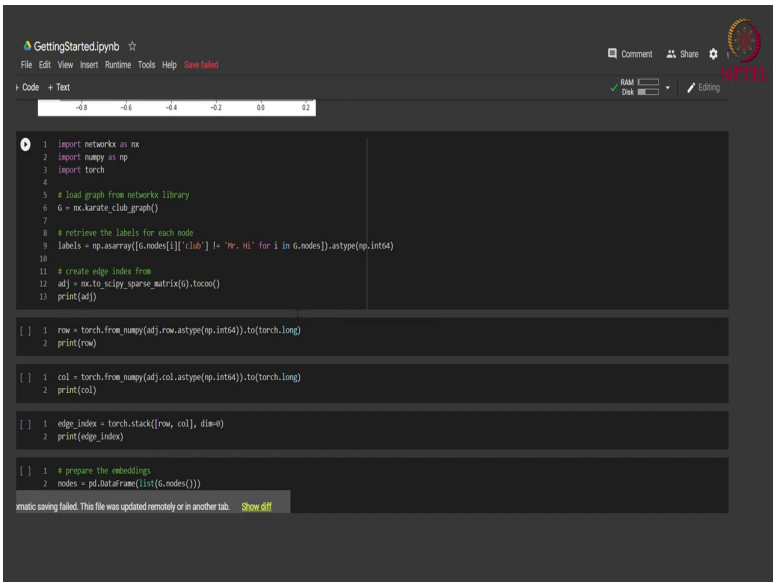
(Refer Slide Time: 27:51)



So, we will use this TSNE plot and we see that we have and you know the color coding that we have done in this plot is basically based on the club of each node. So, we see that this red color nodes belong to one club, you know either mister high or the this blue color belongs to officer and this red belongs to mister high.

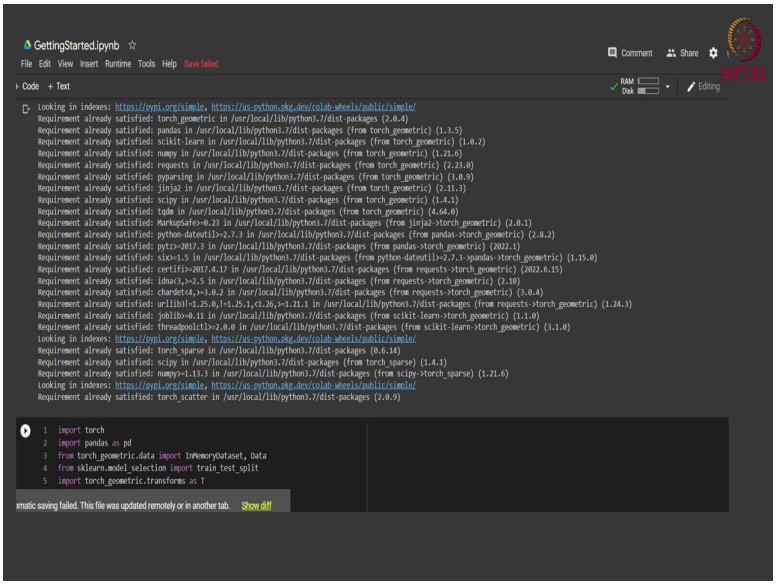
So, these belong to two clubs and we can see that even with the DeepWalk embeddings that we have, these two classes are fairly separable. But, can we you know can we create a bigger margin or can we you know reduce the error in the two in the separation of the two classes?

(Refer Slide Time: 28:35)

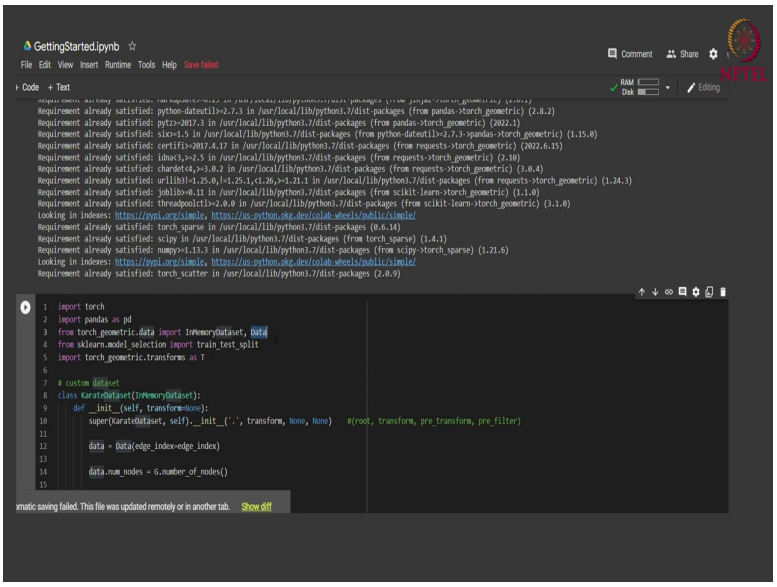


So, we will just we will try to do that with the approach of GCN right. So, again we just ha so, now what we will do is now in order to create the; so, we were talking about creating this data set right from our graph. So, now, in order to create the data set, we will use this library called PyTorch geometric, we will use the same library to implement GCN also. Now, to this PyTorch geometric library, it provides us with a few you know with a an already with a few abstract classes basically, which can be used to implement our own data set right.

(Refer Slide Time: 29:10)



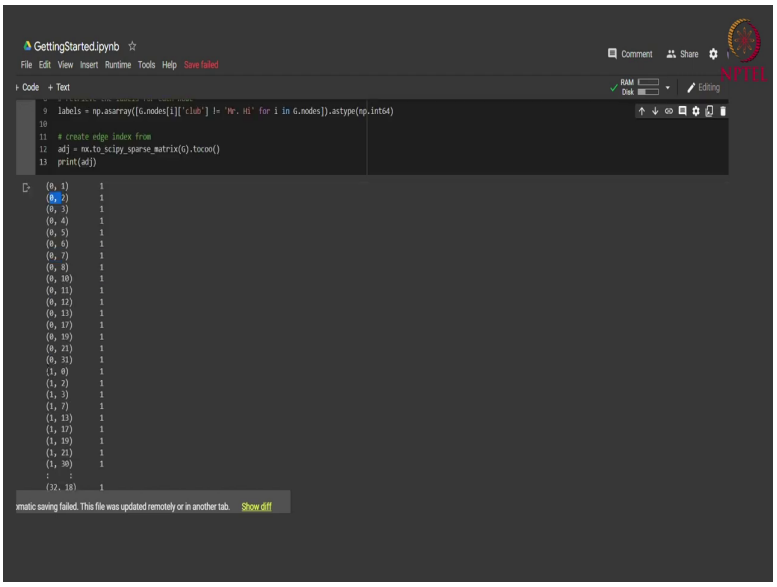
(Refer Slide Time: 29:17)



So, now, this abstract classes which like we are we will be using two abstract classes like in memory data set and data. So, this data class, it basically initializes a graph object for our class right, for our data set. And, this graph object the way it is initialized, it needs a its need like an edge index right. So, this edge index is basically a list of list, where the first list represents the represents the source node and the second list represents the target nodes between which edges are present right.

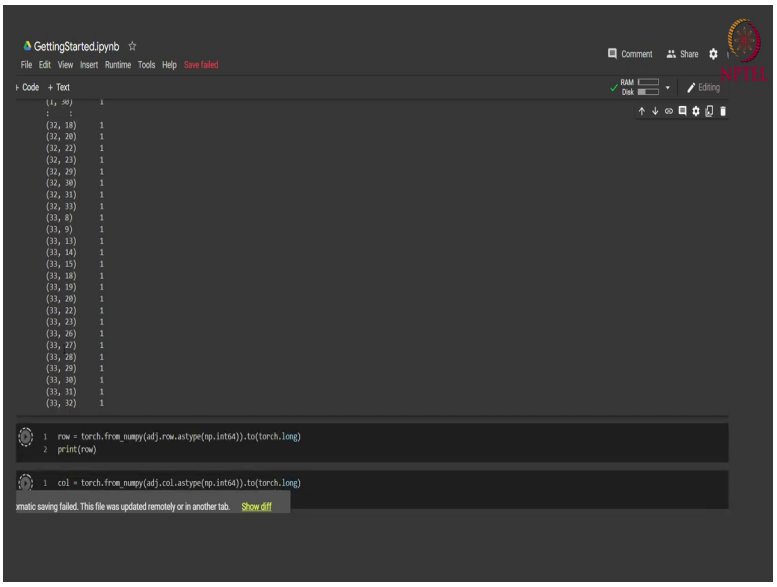
So, if we look at the two inner list element wise; so, there is an edge between the between element wise pairs of the two list right. So, now, in order to obtain such an edge list, such an edge index, we simply use the we simply just first you know first get the adjacency matrix from the graph object using the you know 2 scipy sparse matrix function. And, we see here when the adjacency matrix will print, we will see how it looks like. So, it basically looks like an index which is index on a tuple format which yeah.

(Refer Slide Time: 30:43)



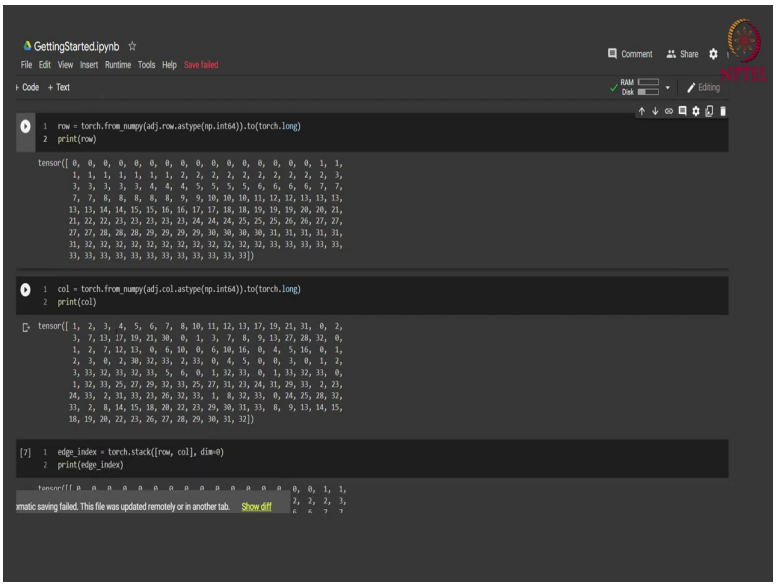
So, here you can see that it is like something like this. So, we have between the source node 0 to 1 there is an edge, between the source node 0 to 2 there is an edge and so on right.

(Refer Slide Time: 30:55)



So, we have this list of tuple indexed you know tuple indexed variable, where this adjacency matrix or you know list it represents something like this.

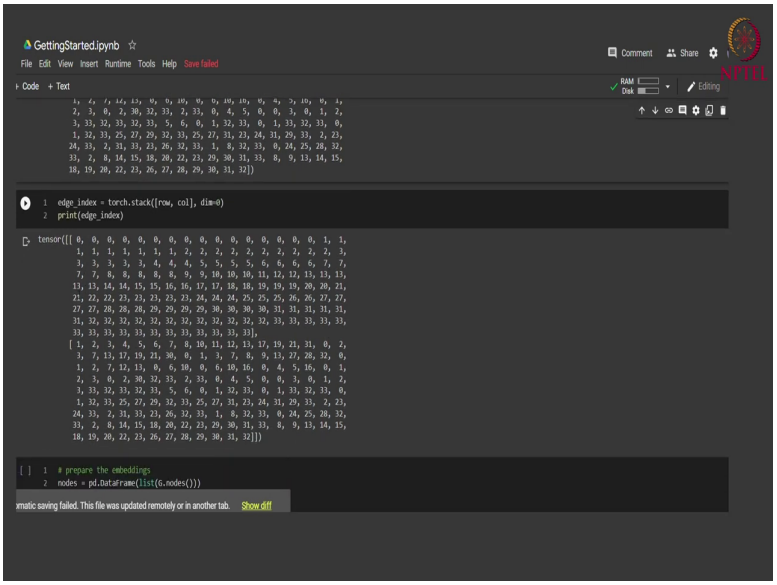
(Refer Slide Time: 31:10)



Then, we you know to create the edge index in the format that we require, what we do is we will extract the rows of all of these you know. So, the basically the rows of the adjacency, the edge index will contain the source nodes right. So, we will extract the source nodes from this couple of you know the pairs that we have by using this you know using this code snippet and we just print this rows. So, we have the first element of each of this tuple.

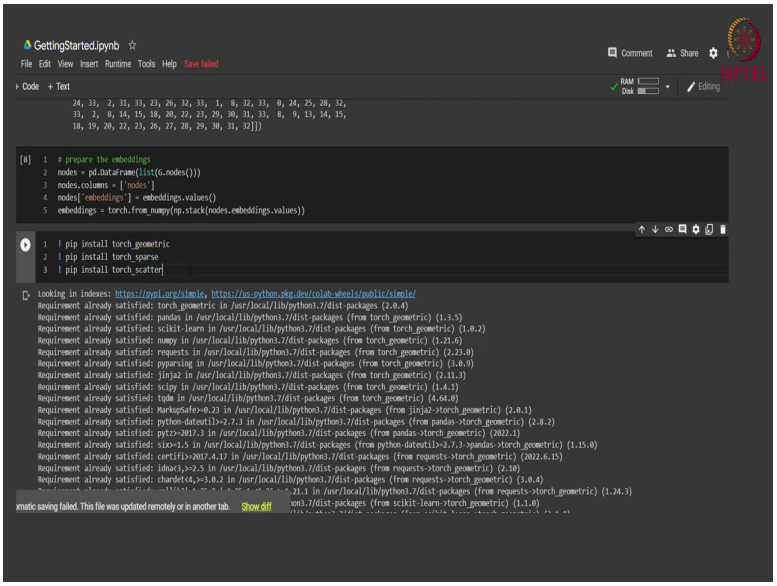
And, similarly we will extract the column that is the second element from each of the tuple and we have something like this.

(Refer Slide Time: 31:49)



Then, we just stack these row and column together to create this edge index in the format that we want. The first list is basically the source nodes and the second list is basically the target nodes right and we just create a tensor out of this.

(Refer Slide Time: 32:05)

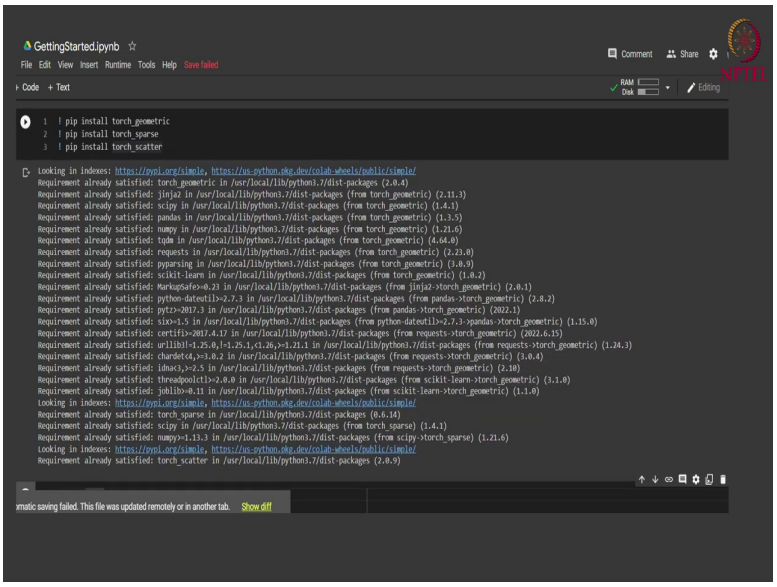


Now, once we have this edge list, what we can do is right and another thing that we want is are the embeddings that we have learned from the DeepWalk mechanisms. So, in order to get those embeddings, we will simply call we will simply have these embeddings or values because if you remember we had these embeddings in a dictionary format right.

So, we just simply call this embedding dot values here and we have this we stack these embeddings one upon the other; so, that we have a matrix of the embedding as well. So, this is basically this embedding matrix of the index for each of the node embeddings. Then, we install some of the essential libraries that we want that is PyTorch geometrics, sparse and scatter. So, these the installation of these libraries it takes some time; so, you better do it beforehand.

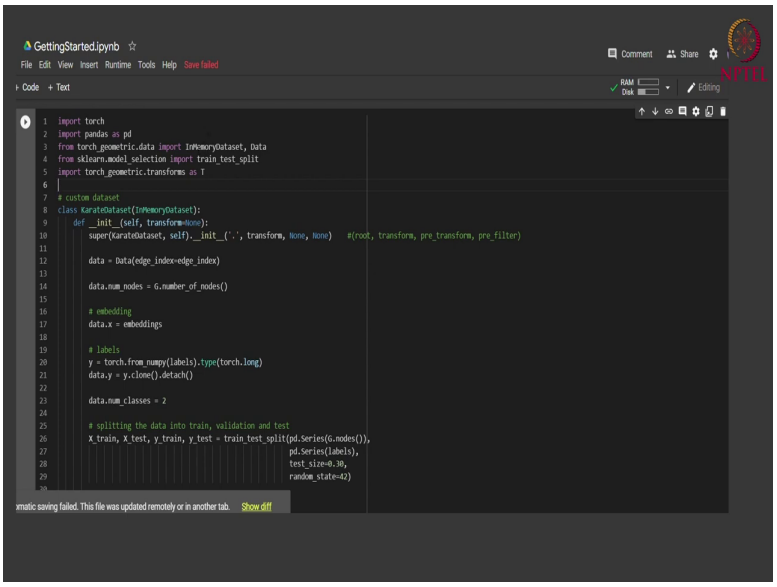
So, yeah so, let it run and we will see if it is able to install right now or not ok so, right. So, after we have these libraries, what we will do is basically we will create our own data set using the abstract classes that we have ok. So, I think our run time is disconnected, yeah it is connected now, let us just quickly run this yeah. So, wait right.

(Refer Slide Time: 34:12)



So, since it takes some time to load these libraries, we have already installed these libraries beforehand.

(Refer Slide Time: 34:22)

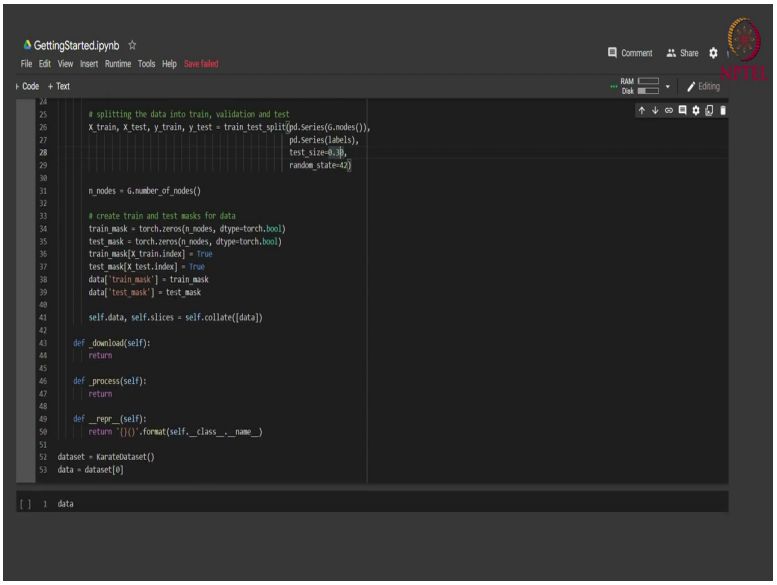


And, now what we will do is we will create our own custom data set right. So, from this torch dot geometric library, we will use these two abstract classes that is in memory data set and data. And, what we will do is we will basically you know initialize our own class and we call it the karate dataset class. And, we have this abstract class that is in memory data set in it, right now we just you know inherit from this class ok.

And, then we just you know simply do the initialization of the super class and so on. Then, for our particular karate club you know karate data set class we initialize the data of it. So, that data is also initialized using this class called data that comes from the PyTorch geometric library and it takes as input the edge index, we have already created that edge index.

So, we give that as the input here and then we you know initialize some other important parameters, that we require such as the number of nodes, the x part of the you know of the data; that is the input that is the embeddings that we have.

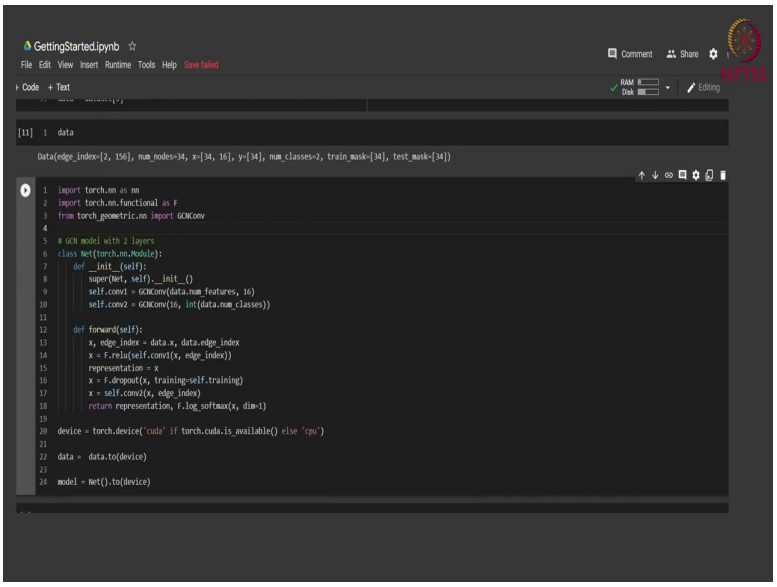
(Refer Slide Time: 35:36)



And, the y part that is basically the labels that we have, that whether the node belongs to you know mister high club or the officer club right. So, we have data y data then x, then we have some number of classes. Since, we have two clubs, we have the number of classes are 2. Then, we simply you know use this use the train test split function from the you know from the sklearn library to get a split of the whole data set into train and test split and where the test side size for our case is you know 30 percentage of the whole data.

And, we create some train and test mass. So, so basically these masks they contains; so, it is a list of you know of size 34 where if a node, if a particular nodes come in the training set that particular value is True or 1 and the rest of it as 0. So, we initialize it as you know all 0s 34 nodes, 34 zeros. And, for each of the you know each of the node that comes in the X train part of the data, we put you know we make the X train mask for that particular index as True or 1 right. So, we have this data datas train mask and test mask.

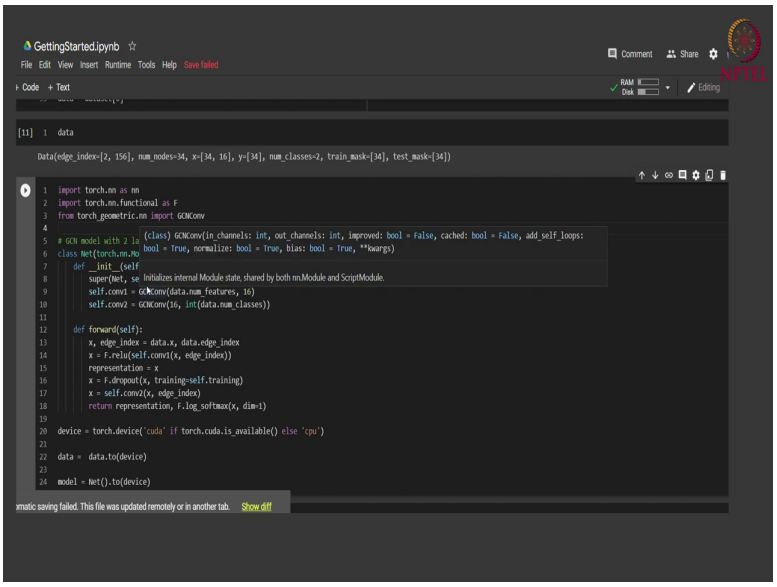
(Refer Slide Time: 36:58)



Then, we have some other function that are already present the abstract class and then we initialize our you know a data set object using this new class that we have initialized and extract the data. And, now if we just print the data, we can see how it looks like. So, this data is basically an edge index, it contains an edge index of you know each of the 2 classes for these number of edges.

And, we have the 34 nodes, where each node is represented by a an with an embedding of size 16 and there are 2 classes and with the train and test mask are 34 size each. Then, we have this no now this torch geometric also provides us with these normal GCN convolution layers that we want right. So, we will just import this convolution layer.

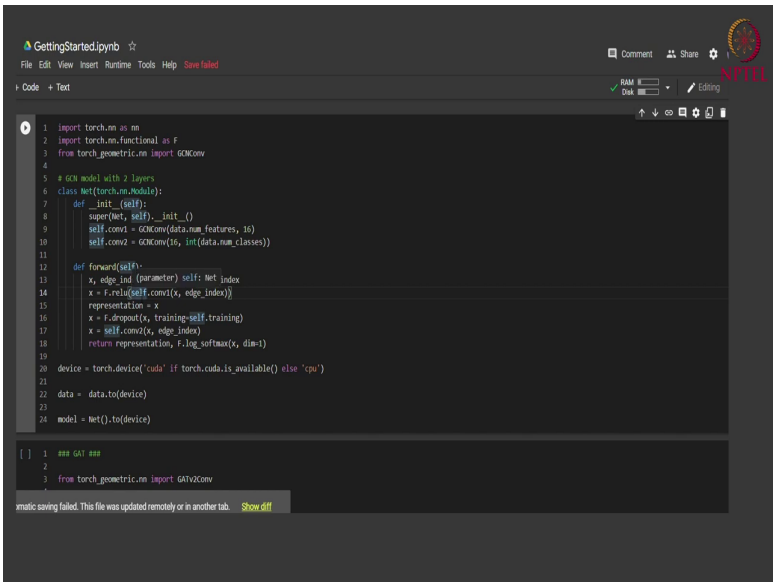
(Refer Slide Time: 37:43)



And, like any other deep learning model that we initialized in PyTorch, we initialize a new class called net which again takes from the nn dot Module you know nn dot Module abstract class that we have.

And, we simply do you know what we simply do is we initialize two GCN layers, one after the other where the first one takes the number of features you know like whatever the embedding size that we have and convert it into a size 16. And, the other GCN like what it does? It takes this size 16 and you know just convert it into the as many number of classes that we have. So, that the final classification can take place.

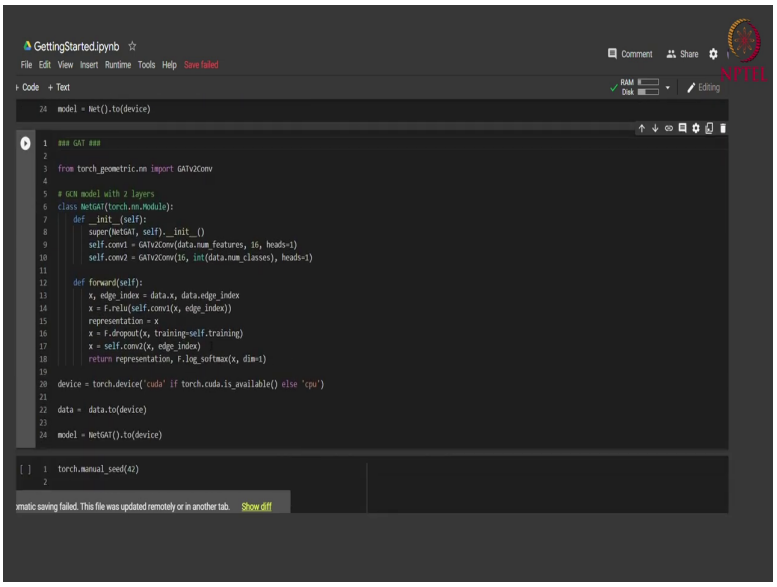
(Refer Slide Time: 38:31)



Now, what we are doing in the forward you know in the forward part of this class is that we have this x and edge index that comes from the data and data edge index. And, we are providing these two x and edge index to this convolution layer, the GCN conv layer which is then the whatever output we are getting from this led is the ReLU activation function. So, that we are you know we are introduced this nonlinearity in between.

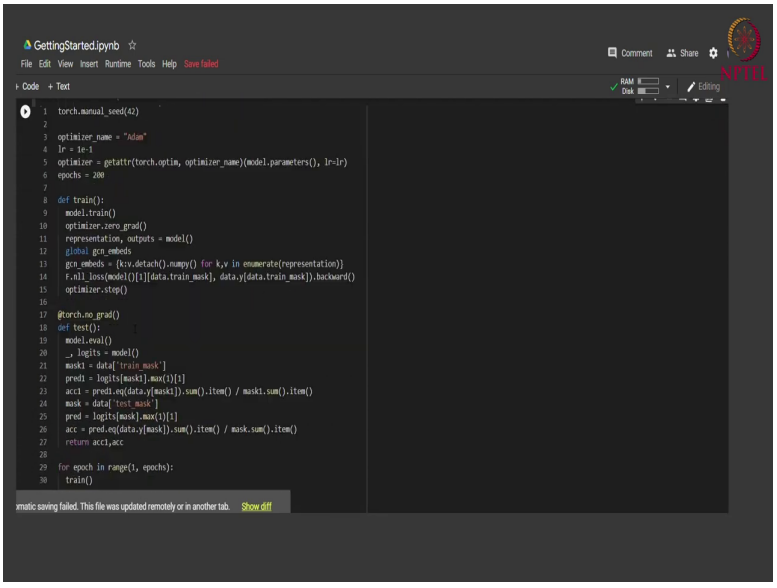
Then, after this first GCN layer, we just you know we extract the representation whatever output is coming after this first GCN layer, we are calling this as the representation that we have. And, whatever output is coming after the next GCN layer that is used to perform this classification. We just you know perform this Softmax over this output and which is further use for classification.

(Refer Slide Time: 39:25)



Then, again we just initialized this Net model and we push it to the device that we want. So, we are here right now we are just using the CPU so, it will stay on CPU itself. Then, we will come to this cell afterwards. Then, we just do normal training and testing of this. So, in the training part we put the model to the training.

(Refer Slide Time: 39:38)

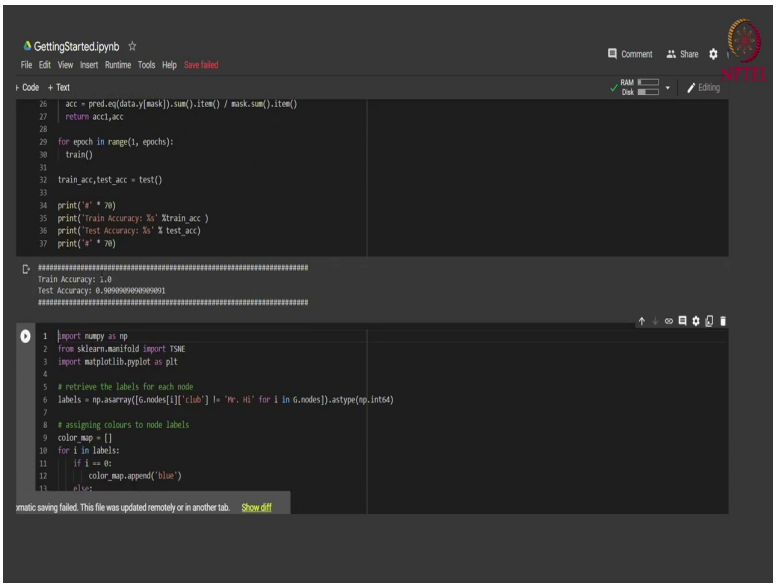


And, then we have this optimizer that we have initialized which is a normal Adam optimizer which basically you know basically optimizes all these model that parameters with the

required learning rate that we are providing. And, for each of the outputs that we are getting from the model object, we have something known as the representation and the final output.

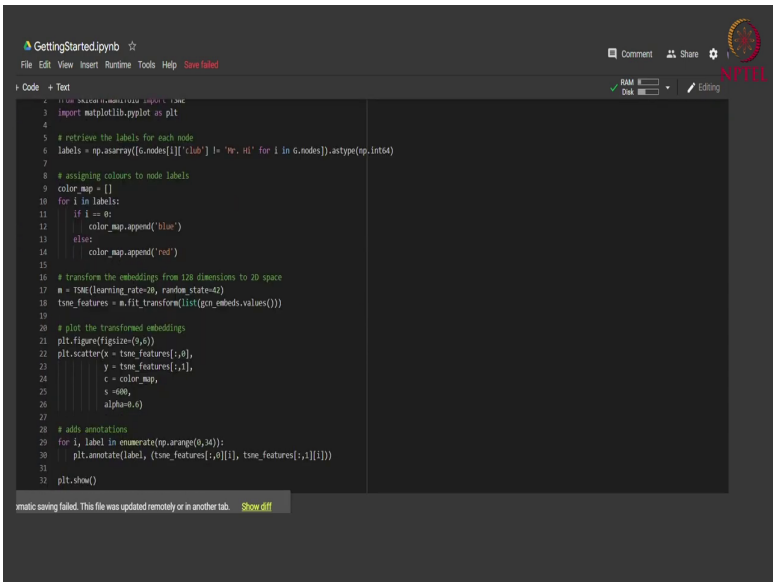
So, for this representation that is the embedding that we have, we are saving it in a variable known as gcn_embeds and we save it in a numpy format right. Then, we have this then finally, we compute the loss between the training you know the ground truth of the training and the representation, that we are obtaining from the from this model. And, based on this loss we are optimizing, we are doing this optimization step that basically you know changes all the weights based on this loss.

(Refer Slide Time: 40:40)



Then, again in the testing part which is calculating the training and testing accuracies that we are obtaining. So, now if you just run it, you can see here that we are getting a train accuracy of 1. So, it is a small data set of only 34 nodes and so, the GCN is able to learn it in a well manner.

(Refer Slide Time: 41:02)



So, we are getting a training accuracy of 1 and a testing accuracy of 0.9. So, now, when you do this TSNE plot; so, initially the TSNE plot that was looking something like this from the DeepWalk embedding.

(Refer Slide Time: 41:16)

Now, if we are plotting this TSNE after the GCN learning, it is looking something like this. So, as you can see the nodes with the same class are now clustered even more together. The last thing that we will see today is the graph attention network. So, that is extremely similar to GCN. So, we have already seen the theory of it, but for the you know for the

implementation point of view torch geometric provides us with a graph attention layer as well.

So, we just simply import this graph attention layers and instead of GCN that we had imported before, we will we simply use this graph attention layers here and rest of the things remain same right. So, we just have this new model now that is net GAT and instead of GCN, we have this graph attention layer. We use the exact same thing, where we you know initialize this model and then the training and the testing loop will remain the same.

And, then again we just you know we plot this and we can see that a similar kind of you know a similar kind of structure is taking place, where the two classes are well separated. So, yeah so, this was all about graph representation learning and how we can implement it in PyTorch and Python. And, we will see you know and I will you know and you should just go and explore more of these methods on your own so, yeah.

Thank you.