Project: Node2Vec

Abhishek Patel (asp15c)

1. Research Problem and Contribution

This project was an implementation of *"Node2Vec: Scalable Feature Learning for Networks"* by *A. Grover* and *J. Leskovec.* This paper deals with learning representations for the nodes of a graph.

In graphs, two main problems being handled by machine learning are link prediction and node classification [1]. The accuracy and effectiveness of the learning algorithm depend largely on the features used for learning. Before the advent of representation learning algorithms, this selection of features was done manually and often required domain specific knowledge. But this is quite limiting and prone to error in selecting features. A solution was to represent each node in a high dimensional space so each node gets represented as a vector. The main objective here is to obtain representations that also depict the relations among nodes, sometimes those that aren't obvious. There are several papers and techniques to learn representations but one method that is quite popular and also used in this project is the skip gram model.

Node2Vec aims to learn representations for nodes such that it reserves the neighborhood properties of each node. The idea of neighborhood is defined in a flexible manner that turns out to be one of the core reasons for the effectiveness of this algorithm. This effectiveness is demonstrated in terms of running a multi label classification task using the representations learned by this algorithm. Hence the goal for this project is to get good results on a multi label classification task based on the representations learned by this algorithm. It should be mentioned here that there already exist good algorithms for learning representations of words in the Natural Language Processing (NLP) area. Although it does not look that obvious for the graphs. But with the help of this paper, the proven and tested method of learning embedding from NLP can be applied to graphs.

As for my contribution, I implemented the core Node2Vec algorithm that generates walks of a graph (details in later sections). I also implemented the algorithm to learn embeddings from walks and finally, the algorithm to perform multi label classification. It should be noted that the paper contains only a brief guideline on the core Node2Vec algorithm to generate the walks. The embedding learning algorithm and the multi label classification algorithm are not a part of the paper. Also the paper contains incomplete information on the exact hyper parameters required to carry out the experiments or any tasks hence these missing values have been chosen and experimented with by me.

One important thing to emphasize here is the separate implementations of embedding learning and the multi label classification task. The combined implementation would roughly halve the time needed to carry the experiment but that would mean that the embeddings learned are for that specific task. Hence I decided to separate them out so that the embeddings are independent of the learning task at hand and that is the objective of this paper too. The experiments in the paper too learn embeddings differently and apply it to just one task to demonstrate the effectiveness.

2. Related Work

There have been several attempts before this paper to learn vector representations for nodes of a graph. This section discussed four such papers. They are each briefly discussed as below:

- **Deep walk: Online learning of social representations -** DeepWalk is an algorithm to learn representations of vertices of a graph. The algorithm consists of two parts, a random walk generator and an update method. Given a graph G, the random walk generator picks a vertex v_i for the random walk's root. The walk then picks a vertex from a set of vertices that are connected to the last visited vertex. This continues till the path length reaches a certain value. This path length value can be fixed for all walks or even be picked randomly for each generation [2]. Now an outer loop runs in the core algorithm, each iteration of which is considered a pass on the data [2]. Then a random ordering of the vertices is created. The randomization proved to make stochastic gradient descent converge faster but other than that has no significance. Then for each vertex in the ordering, a random walk is generated. And then this walk is fed to a skip gram module that adjusts the learned representation based on an objective function [2].

- **LINE: Large-Scale Information Network Embedding -** This paper aims to create embedding for large networks. The algorithm can handle all types of networks (directed/undirected, weighted/unweighted) [3]. The weight on an edge is also called first order proximity of the corresponding nodes and is assumed zero for disjoint pairs. Second order proximity is a function of number of first proximity nodes shared by two nodes [3]. The aim of the algorithm is to create an embedding that preserves the two orders of proximities mentioned above. For preserving the second order proximity, the hypothesis followed is that nodes that have are connected to a lot of common nodes must be related [3]. The crux of this algorithm is the objective functions that the authors picked to optimize.

- **LRBM**: **A Restricted Boltzmann Machine based approach –** While most algorithms consider the links among nodes as a decisive factor in learning representations, this algorithm also aims to take into account the attributes of the node itself. According to the authors, at the time of publishing this paper, this algorithm was the first deep learning method to learn representations from graph data [4]. LRBM stands for Restricted Boltzmann Machines for latent feature learning in Linked data [4]. Each node's representation contains two parts, one which models the sender behavior and another that models the receiver behavior. Sender behavior affects the node's actions on outgoing edges and receiver behavior affects the incoming edges [4]. Building upon this an energy function is created that depends on two tensors that are learned and on the attributes and link properties for the nodes.

- **Distributed Representations of Words and Phrases and their Compositionality –** This paper proposes an algorithm famously known as word2vec. This algorithm maps words to vectors of a high dimensional space such that there is a correlation among the vectors depending on the correlation among the words themselves. It was a very important paper in the NLP field. This paper discusses a skip gram model to learn vectors that represent words and phrases. Another important thing in the paper is the use of Noise Contrastive Estimation (NCE) over hierarchical softmax. This algorithm depends on the following hypothesis: Similar words occur in similar context [5]. The NCE works on the following principle: rather than calculating a softmax probability over the entire vocabulary, you sample a few negative words (wrong answers) that will represent the entire vocabulary and then run softmax on this subset [5].

3. Details of Implementation

3.1 Conceptual Explanation

This section gives an in depth explanation about the algorithms and other implementation details about the project. First the core algorithms will be explained and then the implementation details.

The Node2Vec algorithm aims to target the task of machine learning tasks of link prediction and node classification on graphs. The first problem in this is to represent the nodes of a graph to the learning framework. Obviously representing each node as a set of features is the way to go but then that gives rise to the question as to what those features should be. One option is to pick them manually but that is not scalable across several tasks. The solution is to learn the feature set as well. That's the area of interest for this paper. The idea is to represent each node as a point in high dimensional space where each dimension represents some feature. The goal is to then learn representations that also incorporate the various relationships among nodes, in this case the property of neighbourhoods. The idea of learning vector representations has been well tested in NLP and it gave some great results. So this paper aims to adapt that process to graphs.

So the main idea for Node2Vec is to convert a graph into a series of walks. These walks act like sentences in a word corpus and then we use the Skip Gram model from NLP to learn representations for our nodes. The way these walks are generated is the Node2Vec algorithm. These walks are generated by a procedure called the Randomized Biased Walk. But before discussing that here's a brief outline on the skip gram model and why it was chosen. In a Skip Gram model for each word we try to predict the words in its context. Context is mostly the words around it in a sentence. This works great since it is based on the hypothesis that similar words occur in similar context that captures a lot of the properties of the natural language. In our case with graphs, we treat walks as sentences and these walks are a result of the neighborhood properties of each node. Hence when we write a walk down and treat it as a sentence and each node as a word, using skip grams we end up with a representation of nodes that was created with neighbourhood properties in mind. Hence the objective function is the following equation where we try to find a embedding function f such that it maximizes the sum of log probabilities of predicting a neighbourhood *N(u)* for a node *u* given its embedding form *f(u)*:

$$\max\text{ (f)} \sum_{u \epsilon V} \log P(N(u)|f(u))$$

Neighbourhood of a node can be nodes that are directly connected to a node as well as some node that is not directly connected but shares a similar role. In this context, there are two hypothesis to learn representations based on neighbourhoods, first is homophily hypothesis which states that highly interconnected nodes should be embedded together. Second is structural equivalence hypothesis that states that nodes that share a similar role must be embedded together. So if one were to perform a DFS walk from the node, one is more likely to discover more of the homophily properties as it will expose all the closely interconnected nodes. On other hand if one were to do a BFS walk from the node, they will venture out further and discover a node with similar roles i.e. structurally equivalent (e.g. nodes that are centers of their clusters). Clearly these two approaches are like two extremes and the real world data is likely to exhibit both properties. Hence this paper uses a randomized bias walk to define neighbourhoods and capture a good amount of both the properties. The biased part of the walk gives control over the walk and allows us to fine tune the parameters according to the task as to if we want to capture more of homophily properties or structural equivalences. The randomized bias walk works in the following manner: all the immediate neighbours of the current node are assigned uneven probabilities and based on those the next node in the walk is sampled. This is then repeated with the selected node as the current node and this loops till we have the desired length of walk. The uneven probabilities are assigned based on the following equation:

$$P(c_i = x|c_{i-1} = v) = \begin{cases} \pi_{vx}/Z \; if\; (v,x) \epsilon E \\ 0 \; otherwise \end{cases}$$

Where π_{vx} is the transitional probability of going from *v* to *x* and *Z* is a normalizing constant. This transitional probability is depends on the weight of the edge as well as two hyper parameters that give the effect of a bias. It is calculated with the formula:

$$\pi_{vx} = \propto_{pq} (t,x) * \; w_{vx}$$

Here $\propto_{pq}$ is the bias parameter and w_{vx} is the weight of edge *vx*, *p* and *q* are the hyper parameters that control the bias and *t* is the node that we visited prior to visiting *v*. $d_{t,x}$ is the distance between *t* and *x*.

$$\propto_{pq} (t,x) = \begin{cases} \frac{1}{p} \; if \; d_{t,x} = 0 \\ 1 \; if \; d_{t,x} = 1 \\ \frac{1}{q} \; if \; d_{t,x} = 2 \end{cases}$$

Parameter *p* controls the likelihood of going back to the node where we came from hence trying to limit the walks to local area if set to a low value. It is also called the return parameter. Parameter *q* controls the likelihood of going farther from previous node and tries to explore far off nodes if set to a low value.

These properties of *p* and *q* allow them to be tweaked to give us a control over if we want to explore more of locality (homophily) or far off nodes (structural equivalence). Also this lets the walks capture a little of both properties and hence more closely model the real world data. Apart from capturing more of real world properties, random walks are also more efficient in time and space complexities.

Having listed the Randomized Biased Walk algorithm, we can now define the Node2Vec algorithm. r is the number of walks that start from each node and l is the length of each walk. It is as follows:

```
1. Preprocess the transitional probabilities $\pi_{vx}$ for the graph $G(V, E, W)$ where V are the
   vertices, E are the edges and W are the weights.
2. Initialize walks to empty.
3. for iter = 1 to r do
      for all nodes $u \in V$ do
         walk = Node2VecWalk($\pi$,u,l)
         append walk to walks
4. Run Stochastic Gradient Descent on walks to learn embeddings f.
```

Procedure Node2VecWalk takes 3 parameters, the transitional probability distribution, the starting node *u* and the walk length *l*. It is as follows:

```
1. Initialize walk to u.
2. for walk_iter = 1 to l do
      curr = walk[-1]
      $V_{curr}$ = getNeighbours(curr)
      s = AliasSampler(Vcurr, $\pi$)
      Append s to walk
3. return walk.
```

In the above procedure getNeighbours returns all the neighbours of the current based on the transitional probabilities i.e. based on previous node *t* and current node *v*. The AliasSampler uses the Alias Sampling algorithm to pick a node randomly given the probabilities in constant time. It does so by preprocessing all the probabilities beforehand and constructing two tables which are used to pick a value randomly in constant time operation. This preprocessing step helps in making this algorithm scalable. Also the three main phases, the probability calculations, the walk generation and the gradient descent; each is parallelizable in its own and hence makes this algorithm scalable. In the implementation, only the gradient descent part is parallelized for each batch the rest is kept sequential. The input graph used for generating walks in this implementation is that from bloggers dataset. The bloggers are the nodes of the graph and the following of one blogger to another is the edge. The edges are bidirectional and unweighted.

The gradient descent to learn the embeddings is fairly simple. It comprises of an input layer that takes in IDs of nodes and produces an embedding. These embeddings are randomly chosen initially and are fixed based on the gradient descent. These embeddings are fed in to a simple layer of weights and biases which then try to predict ID of another node. The objective here is to predict nodes that occur nearby a node in a walk. Now since the number of nodes is too high calculating a probability by means of a softmax layer will be too expensive hence a different method called Noise Contrastive Estimation (NCE) is used to measure the loss and gradient descent is applied to minimize that loss. NCE works on following principle, it samples a few nodes from the pool of all nodes and treats them as negative labels; desired node given via training example is the positive label. It then calculates loss in terms of how closely it predicted the negative and positive samples by penalizing divergence from positive label and convergence to negative ones. The training data here is a list of tuples where first element is any node and second element is a node that occurs in a window around the first one in a walk. This is a fairly standard technique used to train embeddings in NLP. It is called the skip gram model.

Finally the third part is to use these embeddings for a task. The task chosen here is that of multi label classification. The dataset used here is of bloggers which contains the groups and its members. The task is that given a node (blogger) we need to predict what labels (groups) they might be in. The dataset is split across a 75% ratio i.e. 75% of data is used to train the classifier and the rest 25% is used to carry out the tests. The classifier model used here is a one vs all classifier. It should be noted that before finalizing this, tests were done using one giant classifier that would try to predict all the labels and also a variant that of one vs all where each classifier would try to predict the probability of the label it represents. A test was also done with a weighted cross entropy loss version of the one giant classifier. These did not produce satisfactory results so the one vs all implementation chosen is one where each classifier predicts a label, i.e. each label has its own classifier and the output is rather a prediction of probability of the node getting that label and the node not getting it. This is conceptually same to the earlier one where it just gave the probability of each classifier but adding one extra parameter improved performance quite a lot. This is because adding even one more additional unit results in several more parameters available to learn and this resulted in a better performance at the cost of increase in running times. So each node is first converted to the embedding which was learned in the previous stage of gradient descent. This embedding is then fed to each classifier which contain a hidden layer of weights and biases. The output of these hidden layer is fed to a softmax layer that tries to predict if that node should get the label or not. A softmax layer basically outputs a probability distribution across all its output nodes, in this case just two; a yes and a no. A maximum of these values is chosen as the decision. This also provides the benefit of assuming that each label is independent and provides an opportunity to make it massively parallel.

3.2 Implementation Details

As for the implementation details the first part, generating walks is implemented in Java. It takes its input from a configuration file. The name of the configuration file is provided as a command line argument. This module has two implementations, one in which it will first generate the Alias Sampling tables for all the edges. At an estimation running that would take a couple days given the size of the dataset. So after several attempts to figure that out a decision was made to generate the sample table on an on-demand basis. This significantly sped up the process but led to a new problem, the number of tables being generated were too huge and were a problem in terms of memory consumption and it also slowed down the performance as the time to perform a lookup for an edge’s table increase with amount of edges’ tables in memory. So another decision was made here to limit the number of edge tables in memory. The upper bound for this is still too high for a modest machine but it can be easily tweaked. The issue with this upper bound is that if after purging a table from the memory if it was needed again, it would be regenerated and so it might be different than its previous formation. This is not really a problem in terms of accuracy as in the end they are used to pick random numbers, but creating them again is a waste of resources. So the reasoning for this upper bound is that given the huge upper bound we have, we are likely to visit all edges nearby to a given edge when table for it was created, hence we reuse the value most of the times. The reuse is possible because for each node we form walks *l* number of times as per the hyper parameter. Hence for a node several edges are bound to be repeated in a walk and hence the reuse occurs during the batch of forming walks for that node. So for a batch of walks in a node they see the same copy of tables, but walks across nodes might see different copies. The performance boost gained by upper bounding the number of tables we store provided stability in terms of memory usage, faster performance at the cost of recreating tables later on. The gains outweighed the cost and hence the decision for bounding was taken. An interesting feature of the implementation is the ability to resume executions, this was initially needed since the program was likely to crash with its enormous memory usage and also for something that runs for hours, so it made sense to create a mechanism to resume operations.

The second module, the gradient descent to learn the embeddings is implemented in Python. It was implemented in tensorflow primarily to be able to use GPU during training. Despite that it takes a couple hours for the most low configuration run and estimated more than a day to run on the highest configuration. For practical reasons I never let it run that long and terminated the runs, the longest I let it run was over 16 hours and that equaled to about 5 epochs on training data on highest configuration. Another motivation for that is that the decrease in loss was too slow to allow it to run for that long, a little tuning of some hyper parameters would have made that rate a bit faster but experimenting on those for something that takes hours to execute didn't make sense. Hence I stuck with some well-known default parameters and others were randomly picked by me and were kept standard across all experimental runs so the comparisons can still be fairly made. Also the paper gives no details on any hyper parameters for learning embeddings and so the absolute performance might vary from the paper's readings. Experiments were performed with varying sizes of embeddings and optimal performance was gained at embedding size of 128 which is in confirmation with the readings in the paper. The input to this module is the set of walks and then a skip gram model is applied to learn the embeddings. On doing a Principal Component Analysis (PCA) of the embeddings for visualizing them, it was observed that the set of nodes were most visibly separated in two groups, smaller groups exists but are not as obvious as these two. This scheme existed throughout all runs of various embeddings sizes which points that there are two main categories of bloggers on an overall basis.

The third module is the multi label classifier. It is implemented in Python and uses tensorflow as well. The model finalized on is a one vs all classifier where each classifier is a binary softmax classifier. The reason for selecting this particular model is already discussed in conceptual explanation section. Each classifier takes as input the node ids which are then translated into embeddings learned in previous module. Then these pass through a simple hidden layer and give probabilities for the node taking on that label in form of a distribution over two nodes, representing "yes" and "no". The paper contains absolutely no information on how to implement a multi label classifier and so the hyper parameters are chosen randomly, so exact results might not match up with that of the paper. Also it was observed that trying to use all of the samples for training led to classifiers learn to always say no i.e. predict 0 probability for the labels, this is because the number of positive samples is extremely low compared to number of negative samples seen in form of the entire dataset. Because of this, the original dataset is sampled to create a ratio positive to negative samples so that number of negative samples don't overwhelm the positive one. This tweak is not that obvious when starting out.

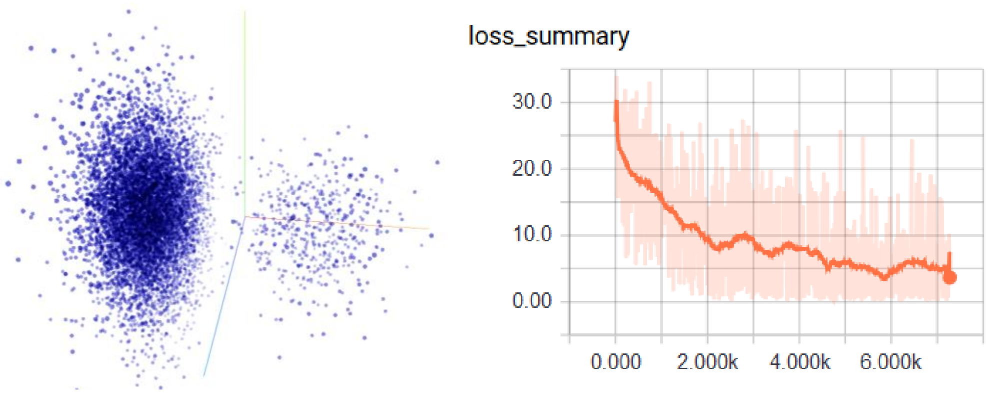
4. Evaluation Methodology and Results

The evaluation methodology is primarily to evaluate the F1 score of the multi label classifier. F1 scores are calculated based on precision and recall. The hyper parameters of the multi label classifier although picked randomly are kept consistent across all experiments. This is in attempt to measure the effect of changing the hyper parameters in walk generation and embedding learning. As mentioned, the experiments take several hours to over a day hence the experiments are not comprehensive of all the tests given in paper. So an effort is made to capture the edge test cases only. Also the tests are not performed for all hyper parameters, for example no tests are done for varying walk lengths or number of walks per node, the window size in skip gram, etc. These parameters obviously increase the amount of data the learning algorithm sees and hence increasing these obviously increase the performance. The F1 scores are calculated in total for each batch, i.e. given a batch of nodes and their respective labels, we calculate the F1 score of containing them as one pool of predictions. Of course dividing them by number of samples in a batch can give per sample F1 score or by number of classifiers to get per classifier F1 score. But it was decided to just add them up so after each batch the F1 score gives the idea of the whole classifier system as a whole. It should be noted that the losses are calculated for each individual classifier, it is only the F1 score that is summed up and that has no effect on the training. So at every step we are printing the micro F1 score and the macro F1 score is calculated at the end of the training. Macro F1 score is defined with average precision and average recall. Now this average over exactly what can be chosen anyway so the method I picked is that of average over number of batches. That is because we calculate precision and recall of a batch as a whole instead of that of individual records of a batch. The paper does not specify if macro F1 is over batches over or each individual record of a batch. The current calculation of macro F1 can be converted from per batch to per record with a simple division by a factor of batch size in calculation of average precision and average recall. Due to ambiguities like these, I decided to stick with one option and carry all experiments on that, this allows for the aim of achieving relative scores instead of absolutely same as that in paper. So, currently we calculate precision and recall of a batch as a whole and then calculate the F1 score.

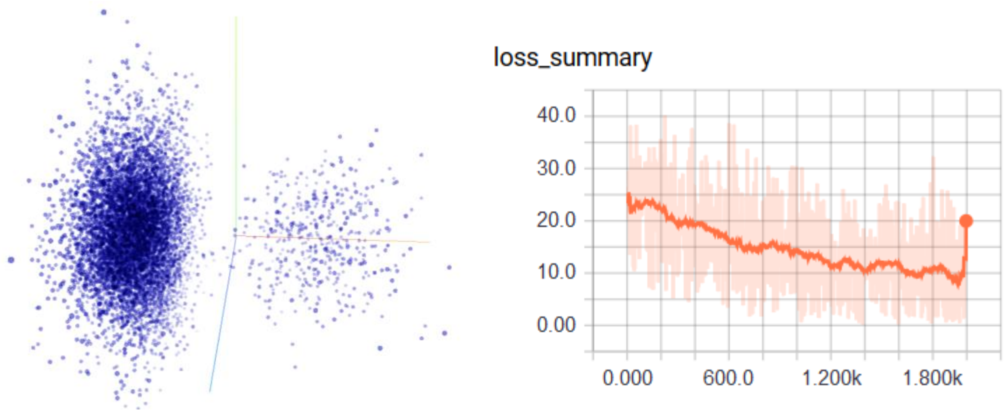
Graphs and readings are as below. The report contains a *fraction* of all the experiments carried out and only contains the most suitable runs that show the effectiveness of the algorithm. There are visualizations on how the embeddings themselves are affected by various parameters. After that are the visualizations of the F1 scores for the classification task. These are micro F1 scores while the paper shows macro F1 scores for some changes and micro F1 for some. Since the paper uses both the F1 variants in different parts, I decided to use micro F1 since they were calculated anyways. Also there is a calculation of macro F1 carried out in the end, it just isn't used in this report; the implementation for it exists and can be found in the source code. Many of these graphs are repeated as each has a contributing property in the best runs.

Embeddings:

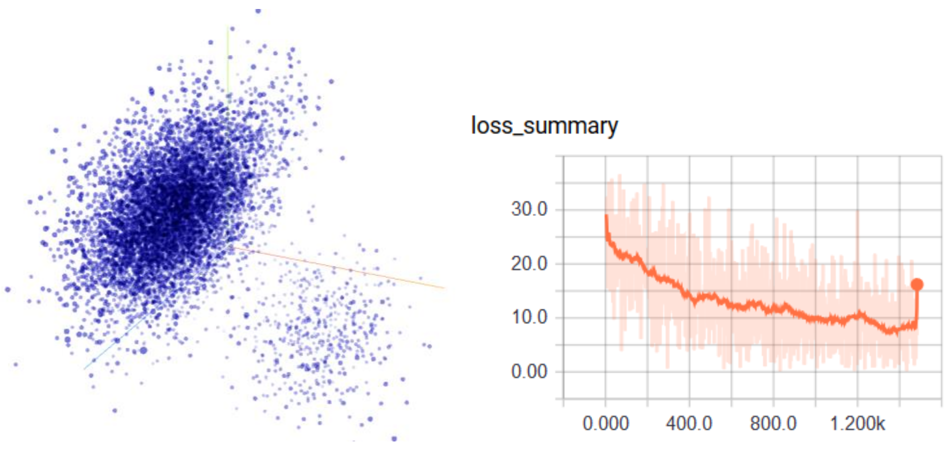
1. For different values of embeddings (p=0.15, q=0.3):



Embedding Size: 16

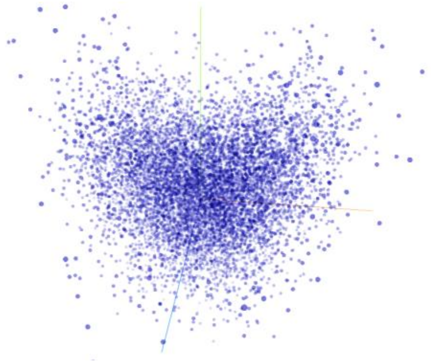


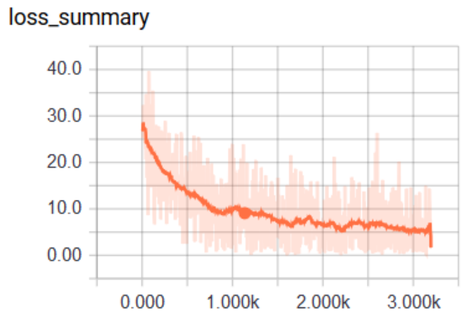
Embedding Size: 32



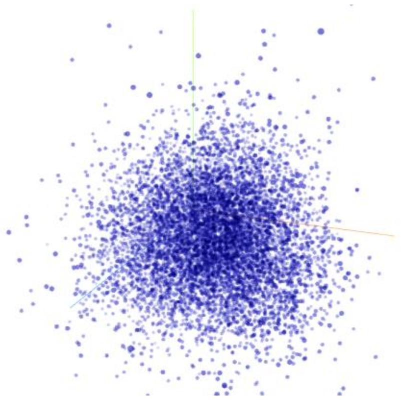
Embedding Size: 128

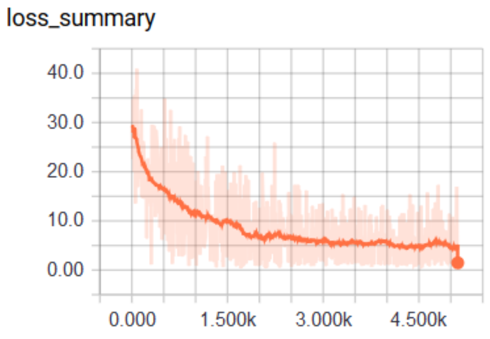
2. For different values of hyper parameter *q (embedding=128):*





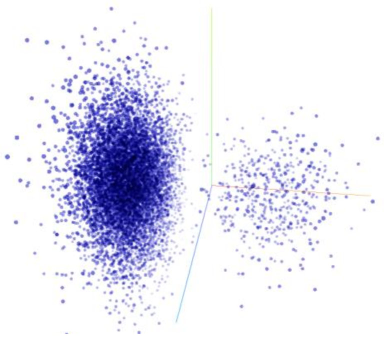
P=1, Q=1

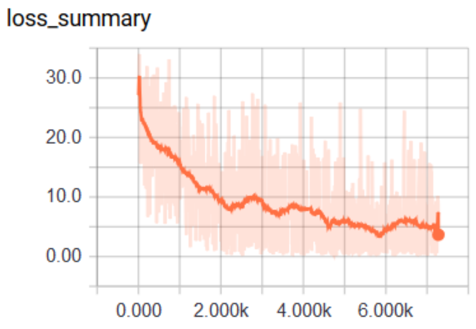




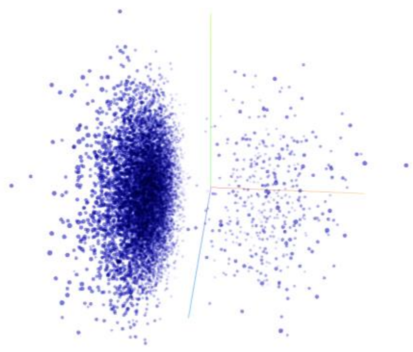
P=1, Q=0.25

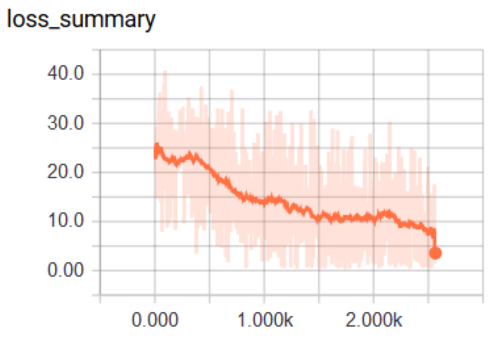
3. For different values of the skip gram's window size:
   3.1 Embedding size = 16





Skip Gram Window = 2

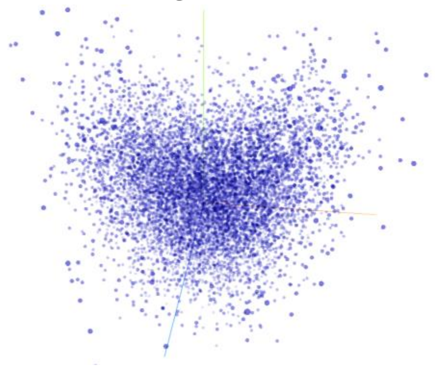


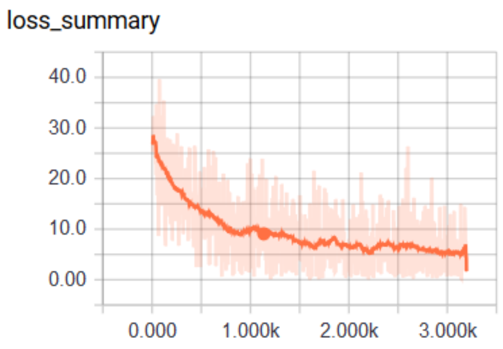


Skip Gram Window = 20

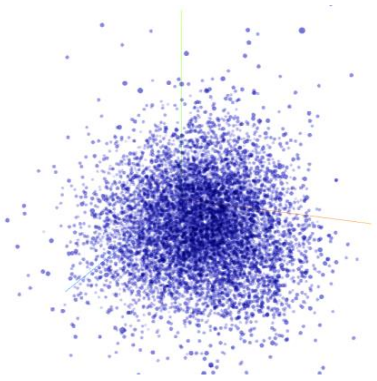
(Here the partitions are neater and well distributed).

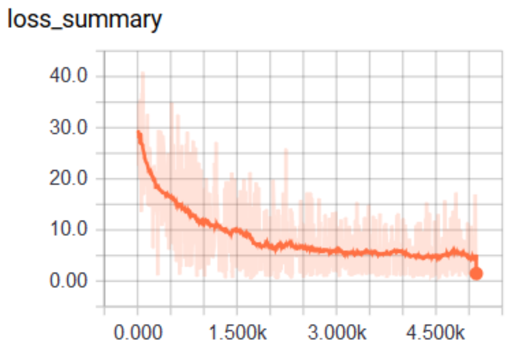
3.2 Embedding size = 128





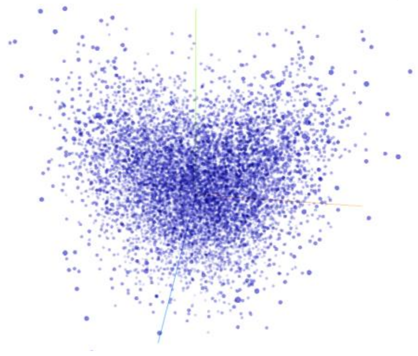
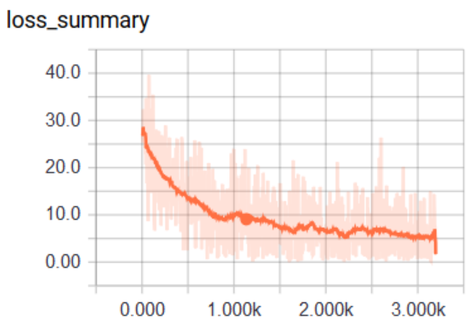
Skip Gram Window = 2. (Sphered View)



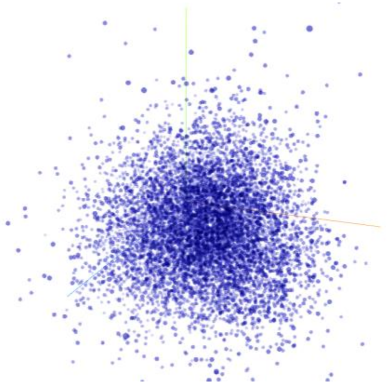
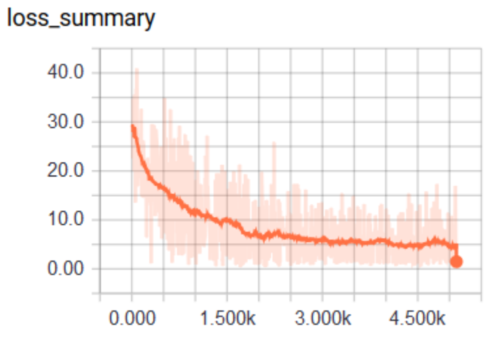


Skip Gram Window = 10. (Sphered View)

4. For different values of walk length and the number of walks per length (*embedding = 128*):

Walk Length: 80, Walks per Node: 16

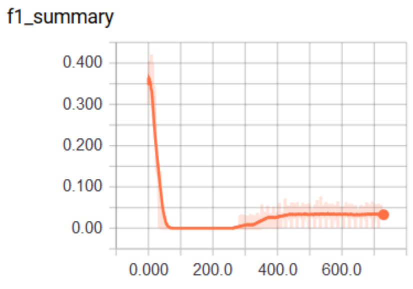



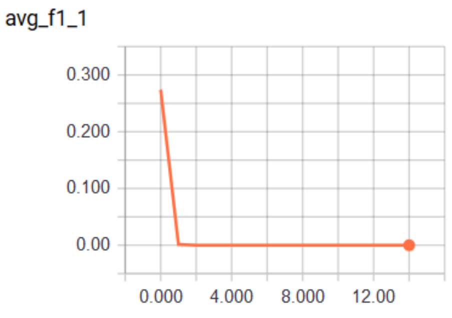
Walk Length: 40, Walks per Node: 6

Classification Task

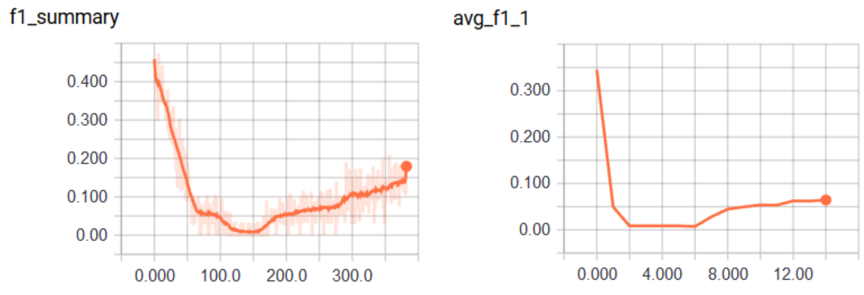
For this section each image pair contains graph for *"f1_summary"* and *"avg_f1"* corresponding to F1 scores of training and validation data respectively. As mentioned before, the whole dataset is split and 75% is used for training, rest is used as validation set.

1. For different values of embed.



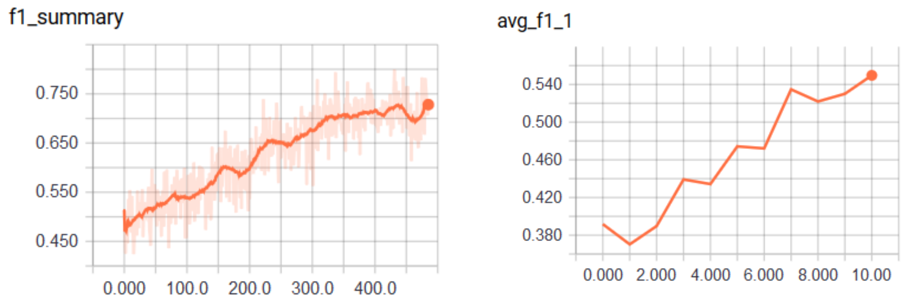


Embedding size: 16

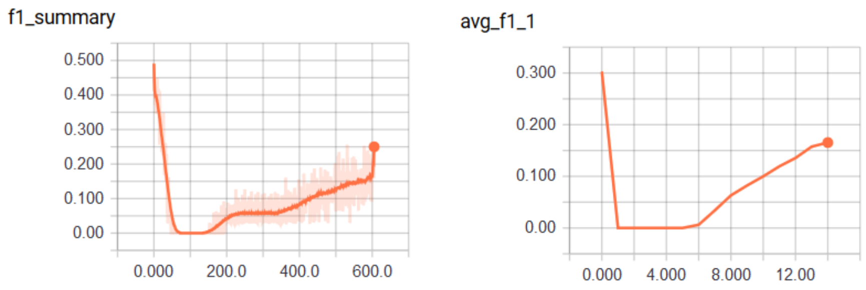


Embedding size: 128

2. For different values of hyper parameter *q*

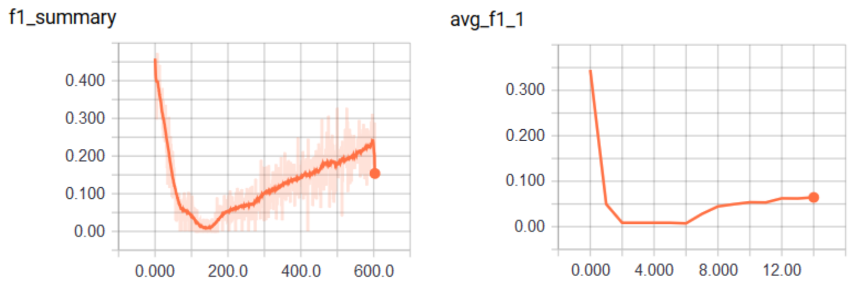


Q=1

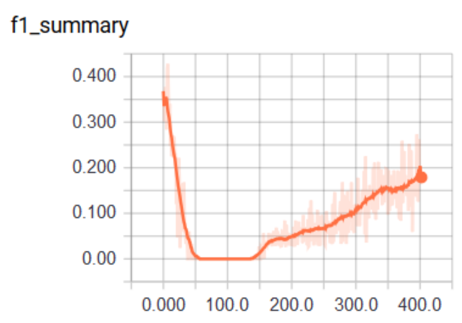


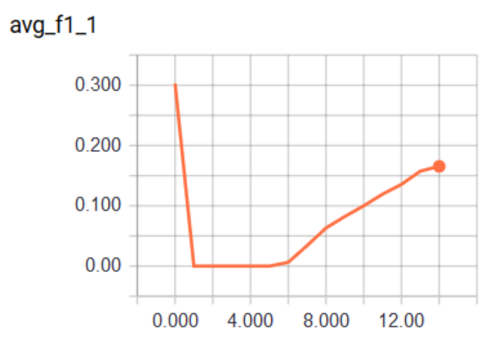
Q=0.25

3. For different values of walk length



Walk Length: 40, Number of walks per node: 6





Walk length: 80, number of walks per node: 16

5. Summary

As can be seen from the results, the implementation although not exactly but relatively followed the same patterns in results as in the paper. The embeddings learned prove to be effective in the multi label classification task at hand. Also one of the achievements for the paper is that the algorithm is scalable and as seen in implementation, it is quite easy to make it distributed. I did not implement it to be distributed because a distributed implementation running on a single machine is meaningless. But tensorflow provides easy library interface to convert existing implementation to run in a distributed environment. Also the Java implementation of the core algorithm can be just as easily parallelized to generate many walks in parallel. Currently it only parallelizes the walk generation and the writing operation.

Also this implementation demonstrates various tradeoffs made in algorithms as well as in their implementation. First tradeoff is in the algorithmic aspect of paper where the balance between BFS and DFS walk properties is gained at the cost of giving away entire gains of either. So while we lose some of the DFS properties, we get BFS properties in return and vice-versa. The balance achieved depends a lot on the hyper parameters of the walk. Second tradeoff was to generate alias sampling tables in an on-demand fashion rather than generating them all beforehand as suggested in the paper. Third tradeoff was in the implementation aspect where the consistency of alias sampling tables was sacrificed for a huge gain in performance. Another tradeoff is in picking low configuration hyper parameters for training to demonstrate the results, i.e. instead of attempting to achieve the exact same results in paper, we just try to get the same sort of relativeness among parameters. This again was for the sake of bringing down the computation time.

To conclude, this implementation successfully models the algorithm in the paper with a few tweaks to improve execution time and resource management. These can be easily removed and with some alterations for distributed environment, this implementation can become the vanilla implementation as per the specifications in the paper.

Links

You can download the project source code and report at the following URL:
https://github.com/Abhishek8394/Node2Vec.

References

1. Node2Vec: Scalable Feature Learning for Networks
   - *A. Grover, J. Leskovec, KDD '16, Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*

2. DeepWalk: Online Learning of Social Representations
   – *B. Perozzi, R.A. Rfou, S. Skiena, KDD '14 Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*

3. LINE: Large-Scale Information Network
   – *J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, Q. Mei , WWW'15 ACM Proceedings of the 24th international conference on World Wide Web*

4. LRBM: A Restricted Boltzmann Machine based approach
   - *K. Li, J. Gao, S. Guo, N. Du, X. Li, A. Zhang, 2014 IEEE International Conference on Data Mining*

5. Distributed Representations of Words and Phrases and their Compositionality
   - *T. Mikolov, I. Sutskever, K. Chen, G. Corrado, J. Dean, Advances in Neural Information Processing Systems, 2013*