Crypto Predicto: Bitcoin Transaction Forecasting with Deep Network Representation Learning

Members: Raghav Pemmireddy, Maximo Moyer, Grant Sterman, Jacob Good

1. Introduction

Our project is an implementation of the paper: Bitcoin Transaction Forecasting with Deep Network Representation Learning. We are using bitcoin transaction data to predict transactions between accounts. We chose this paper because it is an exploding currency and we are curious to use the transactional data given to see if we can predict outcomes. Such results could have interesting implications for fraud detection or predicting price dynamics. This is a supervised learning problem as we know the labels of all transactions and we are trying to train a model to classify them.

1.1. Related Work

Learning embedding of transaction graphs is a cutting edge, and increasingly popular manner of learning about transaction dynamics between entities or accounts. It allows us to mathematically represent the semantic understanding of accounts, which can empower machine learning models, such as ours, to draw conclusions about, or predict actions for these accounts. Capital One [1] implements a similar model to ours, although using a bipartite graph, in order to create these graphical representations of transactions for both accounts and merchants.

2. Methodology

2.1. Data Loading

The first step in our implementation of the paper referenced above is obtaining and parsing the data, available at https://senseable2015-6.mit.edu/bitcoin/. The raw uncompressed data is too large to store on our personal machines, but we create our own data parsing script to scrape the compressed data and format the first 100k sender-receiver pairs for transactions in the Bitcoin Blockchain Data into a weighted edge list. The inputs to our script are a file containing information about transactions, and 2 other separate files containing addresses which match inputs and outputs, as well as amount sent/received, to their specific transactions. For each edge, input address is the source node, output address is the destination node, and transaction amount is the weight. We save these to an output file which we used to create graphs in our next step. Our statistics concerning the number of unique accounts and transactions in our edge lists are closer to the paper's implementation when we count unique sender-receiver pairs and do not consider uniqueness in both directions. (Ex: 1 => 2 and 2=1 are two unique pairs.) Refer to 4.1-Challenges for more information.

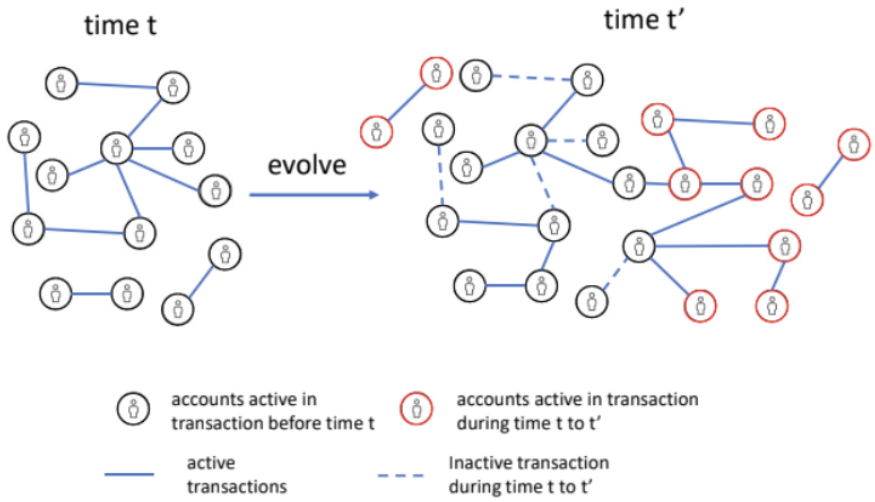

Figure 1: Graph Creation Visualization

2.2 Creating Graphs: Preprocess.py

Once our edge lists are formed in Pre-preprocessing, graphs can be created. In accordance with the methodology laid out in the paper, we create a new graph at each timestep (t=0,...,9), which is created with 10k unique send-receiver pairs. For each new edge added to the graph, we add a weight to the edge and set it equal to 1. We then take the graph from the previous timestep, decay the weights of all edges by .5, and add all nodes to the new graph that have one edge with a weight >.125. This implies that every time a node does not have a new edge added to it over 3 timesteps, we delete the node. This is done because of the dynamic nature of bitcoin accounts. Account numbers often change users, thus when an account goes inactive in the short term, it is crucial to delete it from the graph, as its next transaction after being dormant may be from a different user - rendering what we previously learned about that account useless. By partitioning these graphs to account for the dynamic nature of bitcoin account numbers, we can train our model's embeddings in Node2Vec in a way that is meaningful.

Timestep (t)	**Number of Nodes in Graph**	**Number of Edges in Graph**	**Number of Nodes Deleted from Last Graph**
0	8308	9992	0
1	14489	19969	0
2	23494	29877	0
3	23875	29846	8232
4	23750	29893	6105
5	21281	29960	9428

6	20486	29990	7733
7	21957	29966	5862
8	22072	30011	6874
9	21520	29943	6907

Figure 2. Graph Statistics at each timestep

2.3 Model Architecture

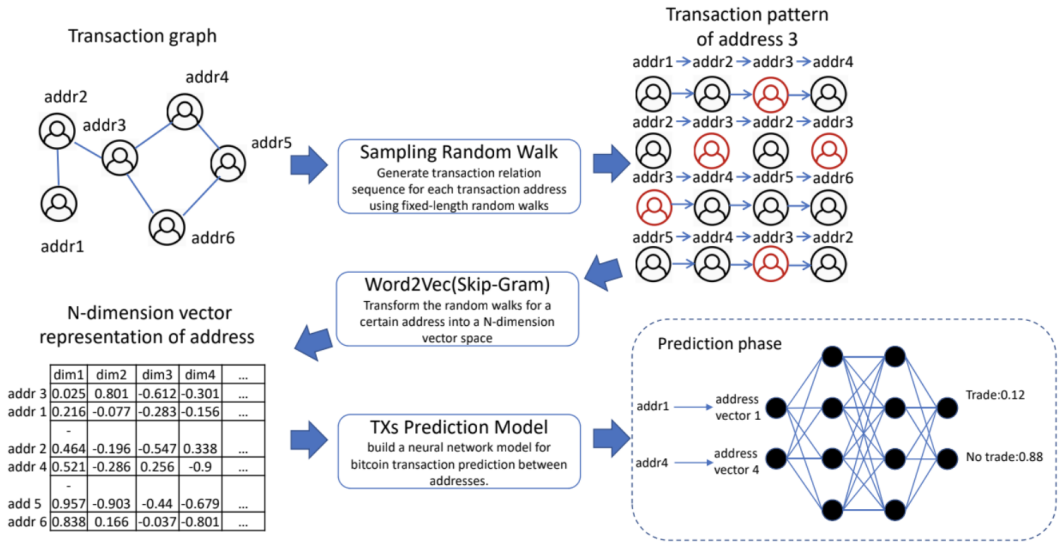

Figure 3: Model architecture from original paper

2.3.1 Node2Vec

We employ a word2vec model to derive our embeddings for the graphical representation of transactions. As in the paper, for each node in a given graph at timestep t, we create 10 random walks of length 40, each assigned a weight of 1. For each timestep t, we decay the random walks from the previous timestep (t-1) by .5 if the walk ends on node u and node u is the destination node for a new edge. We then delete each walk if their weights are less than or equal to the threshold of 0.125. We append all random walks with acceptable weights to our newly created walk list, and call these the walks for timestep t. This decaying process is meant to prevent the overrepresentation of nodes that occur in many transactions. After we get our random walks, we create our vocabulary dictionary of nodeID to embeddingID, and we format our input data using skip gram sequences for each walk using a window size of 1. We began with a window size of 2 but found that a window size of 1 was the only way for the model to run within memory limitations of personal and department machines. We then train a language model using the skipgrams with one embedding matrix and one linear layer for one epoch while using batch sizes of 1000. We utilized sparse categorical cross entropy loss with logits to update our embedding,

weights, and bias matrices. Next, we turn these embeddings into a dictionary of embedding ID to nodeID so that we can match up our embeddings to account IDs in our prediction model. In attempts to decrease runtime for the entire model, we save our embeddings, embeddingID to nodeID dictionaries, and edges for each graph. This allows us to have flexibility of running our predict model without being dependent on the Node2Vec output in realtime. We intended to run this for all timesteps, but creating embedding matrices proved to be incredibly time-consuming, and increasingly so at each timestep. For timestep t =7, our embedding training took 24 hours to run, thus we were only able to create embedding matrices for 8/10 timesteps.

2.3.2 Predict

Once embeddings are learned in each graph, we can take pairs of accounts (represented by their embeddings) and forecast whether transactions happen between those two accounts. The architecture of this model is as follows. At timestep t, we first load in the embedding matrix, graphs, and a dictionary that maps NodeIDs to an index for the time t embedding matrix (trained on random walks from time t). We then use graph t to return a list of the transactions between nodes in a graph (in the form of concatenated embedded src and dst nodes). This is then concatenated with a list of an equal amount of randomly selected negative samples (that take on the same form of concatenated embedded src and dst nodes). We also return labels corresponding to these returned transactions indicating which transaction occurred and which did not. Negative sampling is done because of the vastly higher amounts of negative labels than positives. We do negative sampling in the form of data balancing, which is our interpretation of the paper's implementation. We tweaked the hyperparameter of negative samples/positive samples to dictate how many negative samples to take and find that the ½ ratio created the most accurate outcomes.

We then train our model on these inputs and labels in a batch format with size 10. This was found to be an effective size for fast training speed. Our model architecture in predict is 3 dense layers with relu activation, with the output in softmax form so we get the probability of a transition occurring or not occurring between the two nodes. Our loss function is categorical cross entropy loss.

We then test our trained model on graph t+1. We do so by creating a list of all node pairs from graph t which exist in and have transactions in graph t+1. We find the embeddings of those nodes to make their features usable inputs. We then concatenate this list of already concatenated node pairs, to a list that takes the same format for an equal number of negatively sampled node pairs (pairs that do not have an edge), also from graph t+1. We then test the data using our trained model. We do this for each timestep where embedding matrices were produced, training on graph t = x, and testing on graph t = x+1, we do this for t = 0,....,6. We measure success using accuracy and F1 score.

3. Results

Timestep (train-test)	**Train Size**	**Test Size**	**Accuracy**	**F1**
0-1	19984	32	.78	.72
1-2	19960	102	.89	.89

2-3	19840	78	.88	.88
3-4	19906	132	.89	.88
4-5	19968	268	.79	.75
5-6	19912	374	.79	.75
6-7	19912	390	.77	.72

Figure 4. Data Sizes, Testing Metrics across Timesteps

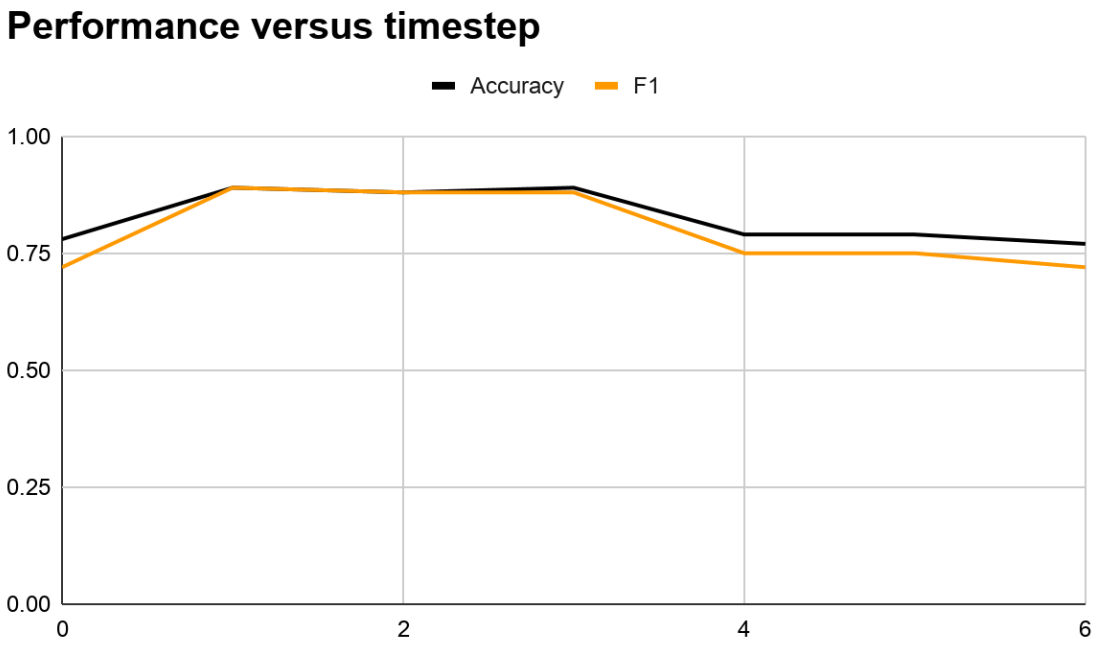

Figure 5. Testing Accuracy and F1 across timesteps

3.1 - Results Discussion

Our results were surprisingly good. We had accuracies and F1 with a means of .83 and .79 respectively. This means that our model was both successful in identifying the presence of transactions versus no transactions in general, and had strong precision (not having false positives) and recall (identifying all positive instances of a transaction). It is important to note that our accuracies and F1 seemed to start decreasing after time step 3. Although we train one prediction model across all timesteps, an important distinction between our model and traditional DL models is that we generate new embedding matrices at each timestep. Thus, this decrease in performance at later timesteps is not entirely problematic, however it is notable, and we are curious if this accuracy would continue to decrease if we had time to train and test future timsteps.

These results were better than those in the paper, where the means were .62 and .69 for accuracy and F1 respectively. These are surprising results, and setting aside any error in implementation, we believe such successful results occurred as a result of how much data we trained embeddings on. In the

paper, a sample was taken from the walks to train embeddings at each timestep, while we trained on all skipgrams generated from walks. This may have led our model to have a better understanding of the dynamics of the graph, and thus lead to higher predictive accuracy. Lastly, it is worth noting that our accuracy was consistently higher than our F1 score, while the opposite was true in the paper. We are unsure why this may have occurred and might have a better understanding with more training/testing timesteps. For more, see 4.3.

4. Challenges and Further Improvements

4.1 Data Loading

Due to our large data size and a lack of expertise on C++, we were unable to effectively run a script found at https://github.com/dkondor/txedges which could've created the edge list for us. This script would have automated the data loading process and given us the edges in the correct format with the same statistical properties as the paper. However, we found that our python script effectively loaded in the data for the first 100k unique sender-receiver pairs, the only problem being that there were discrepancies in the statistical profile of our data (number of transactions, number of aggregate accounts). We were not sure how to solve this problem but believed that it would not be a huge issue in terms of our model's functionality given the discrepancies were minor, and that we still had a similar amount of data although the exact data used may have slightly differed.

4.2 Node2Vec

The methodology of training node embeddings in the paper took orders of magnitude less than ours. Our model takes 4-24 hours per graph to train embeddings but converges to a general range of loss after 20-30% of training data. As a result of the long time taken to train our Node2vec model, we were unable to try training on a sample of skipgram data instead of the complete data but hypothesize that a model trained on sampled data would have performed no better than ours.

Additionally, the paper cited multiple different other papers with differing methodologies on how to learn node representations through random walks, and we were unable to pin down an established algorithm, such as DeepWalk[2] , which we could reference. We therefore think that we might have created data in a different way than the methods used in the paper, impacting our training time for learning node embeddings.

4.3 Prediction Data Size Challenges

Because we calculated our accuracy on transactions that are present in consecutive graphs, our data sizes for predictions were relatively small compared to the overall number of transactions. For predictions from timestep 1 to timestep 2, our model only identifies 16 transactions which carry over between graphs, leading to relatively small test data of size 32 after negative sampling. This size of test data became larger over time, approaching the mid 300s towards the later timesteps. The paper was unclear about this issue and how to address it so we assumed this was a result of their implementation as well for the prediction model.

Additionally, we balanced our data so that there were equal labels for both 1s and 0s in each of the train and test data. This was due to our interpretation of the paper's sampling methods, and it could have played a role in the relationship between our accuracy and f1 measurements.

4.4 Other Graph Construction Methodologies

The original paper that was the inspiration for the project also implements the prediction model we did for the last 100k unique sender receiver bitcoin pairs. They did this as the nature of bitcoin transactions changed over time as bitcoin became more popular. This lead to more accounts being involved in one transaction, and less money being transferred per transaction. Thus, a reachability model (the model we implemented) is also implemented on the later 100k pairs to see if it is still an accurate transaction prediction method for the more recent transaction graphs which have different characteristics.

Moreover, the paper also implements a prediction model that uses the amount being transacted between two accounts at time t to predict if transactions in the future will occur. It does this both for the first 100k and last 100k sender receiver pairs. This prediction model has similar accuracy and a similar F1 score to the reachability model in the paper.

Finally, the original paper also implements the same techniques using a static graph to train embeddings. This results in less accuracy and lower F1 score because it does not take into account the dynamic nature of bitcoin transaction graphs.

Given the scope of the project, we did not have enough time to implement these additional models or implement our model on the most recent 100k sender receiver pairs.

5. Reflection

We felt our project turned out very well. We beat all of our goals, which were centered around accuracy, given that we actually beat the paper's implementation. In every facet of our project, there were many moving parts outside of the basic implementations of skipgrams and feedforward deep learning models. We found that our project required extensive knowledge of blockchain and bitcoin, which resulted in significant effort in understanding the paper before we even tried to implement it. Most of our time in writing code was spent in pre-processing and data loading, as in those processes, we had to come up with algorithms to quickly create graphs and random walks (which was difficult given the size of the data). In running our model, the extensive time taken to train node embeddings hindered our ability to tweak many hyperparameters and explore other methodologies of creating random walks and data. If doing the project again, we would like to find a way to more efficiently create graphs and walks to allow us to tweak our hyperparameters in order to maximize accuracy. In our own implementation, one pivot we had to make is that we originally intended to implement the static and amount prediction methods from our paper, but quickly realized we would not have time to do this.

Our model generally worked as anticipated. We followed the RandomWalk, Word2Vec, and Prediction model that the paper implemented. However, we did have to come up with our own method of creating walks which proved difficult. One unexpected challenge was that we lacked testing data due to the lack of persistence of accounts transacting from one graph to the next. So, we had to make a judgement call to balance negative and positive data as the paper did not specify this amount.

Given more time, it would have been interesting to explore new sets of blockchain data other than the first 100k sender-receiver pairs or implement the amount graphs or static graph methodologies. Given that our accuracy was higher than that found in the paper, we are curious if this was due to luck, an error,

or a superior aspect of our implementation. We suspect this higher accuracy is due to how we train node embeddings using all data instead of a sample. This is a double-edged sword due to the increase in training time which makes the model hard to implement in real time situations such as fraud detection tasks. However, at the same time, if we do have higher accuracy than the paper's authors, this would be a significant accomplishment, and we hypothesize that we could validate our greater accuracy by testing our model on amount and static graphs that we would like to implement in the future.

Our biggest takeaway from the project is that when abiding by an overarching model (i.e. using Word2Ve) there is more flexibility in implementation than just tweaking hyperparameter. We saw subtlety in the way you can create the original transaction graphs, random walks, and select training and test data. We believe the decisions we made around these three topics had a profound impact on our results. Finally, we also learned that real-world preprocessing is messy!

Sources

[1]:

https://www.capitalone.com/tech/machine-learning/learning-embeddings-of-financial-graphs/

[2]:

https://arxiv.org/abs/1403.6652

[3]:

http://www.perozzi.net/publications/14_kdd_deepwalk.pdf