ORACLE

Automatic Graph Expansion and Graph Machine Learning

Miroslav Čepek

Data Scientist at Oracle Labs, PGX

Agenda

- Financial Fraud Case Graph – Automatic Expansion
- Graph Machine Learning, Node and Graph Vector Representation
- Contest Definition

Financial Fraud Case Graph – Automatic Expansion

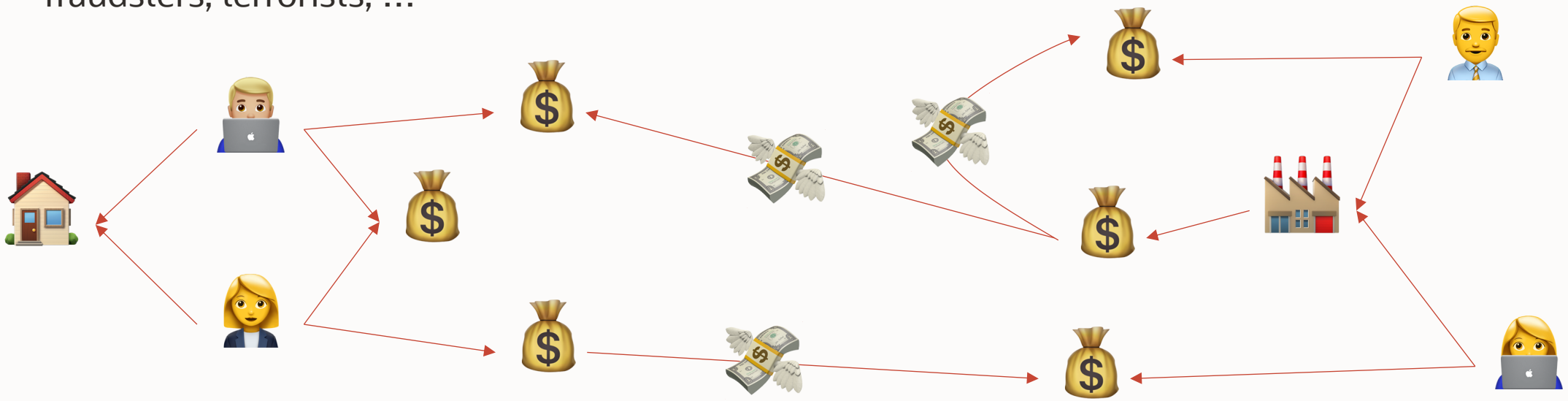
Bank's operations data – what you can look at

- Customers
- Address
- Internal Accounts
- External Account
- External Persons and Companies

- Various types of money transfers
- Account Ownership (private or joint accounts)
- Address of residence, corresponding address

Financial Graph

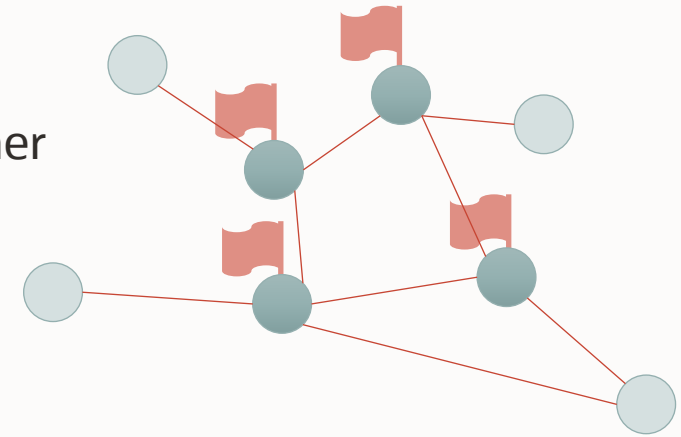
- Bank's operations can be visualised as a graph.
  - customers, accounts, external accounts are seen as nodes.
  - financial transactions, account ownership, known relationship are seen as edges.
- Operation data are often supplemented by external data – company register, list of known fraudsters, terrorists, …



Financial Case Graph

- Bank is obliged to detect a financial crime.
  - Detect potentially problematic behaviour and report it to proper authorities.

- There are simple rules to flag problematic entities and transactions.
  - If flags accumulate around specific node(s), it and flagged transactions and nodes are turned into a case.
  - Case is manually investigated by an analyst.
  - A part of investigation is to expand the initial case graph with nodes that are not flagged but are relevant to the case.
  - The identification of relevant nodes (case expansion) is crucial to the correct decision whether the present case has legitimate explanation or is case of financial crime.



Graph Machine Learning and Graph & Node Embedding

Graph Machine Learning

- Aim is to perform machine learning task(s) on data represented by graph.
  - Classification and clustering of individual nodes in the graph
  - Classification and clustering of whole graphs

- *Identify role of a computer in the computer network from it's network traffic.*

- There are some non-ml techniques, metrics to look at – betweenness centrality, pagerank, …

- So far, there are no native graph classification/clustering techniques.
- The approach is to build vector representation of nodes or graphs
  - Node embedding
  - Graph embedding

DeepWalk

- Approach based on co-occurrence of node types in the neighbourhood.
- Vertices with similar neighbourhood have similar vector representation.
- Strongly inspired by NLP Word2Vec.

- The vertex neighbourhood is explored by taking random walks and noting down nodes the walk visits.

Random Walk

The idea of random walk is simple.

1. Start in a random start node V_0 and set $\mathrm{V} := V_0$. Set list of steps to $W = [V_0]$.
2. List all edges going out of V, follow randomly selected edge to the neighbour N.
3. Append selected neighbour to steps in the walk $W = W + [N]$.
4. Set $V = N$
5. If length of the walk W is longer than max length parameter k, finish the walk. Otherwise return to step 2.

DeepWalk Ideas

1. The first idea, how to leverage information obtained from random walking around the graph to estimate $P(V_{i+1}|W) = P(V_{i+1}|V_i, V_{i-1}, V_{i-2}, \cdots, V_0)$.
   - As important improvement, you can cheat and look forward in W as well as backwards - $P(V_i|V_{i+L,} \ldots, V_{i+1}, V_{i-1}, \cdots, V_{i-L})$
2. It turns out that predicting $P(V_{i+L,} \ldots, V_{i+1}, V_{i-1}, \cdots, V_{i-L}|V_i)$ is much easier.
   - Basically predicting neighbouring nodes in the walk, based on the present vertex.
3. Instead of actual vertex, generalise to vertex representation $\Phi(V_i)$
   - $P(V_{i+L,} \ldots, V_{i+1}, V_{i-1}, \cdots, V_{i-L}|\Phi(V_i))$
   - Nodes with similar neighbourhood have similar representation.
4. Remove constraint on ordering, assume indecency and estimate $P(V_{i+k}|\Phi(V_i))$.
   - where $k \in \{-L, -L+1, \ldots -1, 1, \ldots L-1, L\}$
   - In other words, assumption is that the probability of observing node V_j in neighbourhood of V_i is not influenced by any other node in the neighbourhood.

DeepWalk algorithm

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
  $\mathcal{O} = \text{Shuffle}(V)$
  for each $v_i \in \mathcal{O}$ **do**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, t)$
    SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
  end for
end for

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

for each $v_j \in \mathcal{W}_{v_i}$ **do**
  for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
    $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
    $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
  end for
end for

- Repeat γ times a random walk from each vertex.
- In each repetition start a random walk from every single node. Order of random walks are shuffled.
- After every random walk, vertex embeddings are updated using gradient descent.

Node Likelihood and Objective Function

- The representation of a vertex $\Phi(v_i)$ is a lookup table (a map) assigning a vector to given vertex.
- The representations are updated using stochastic gradient descent, gradient with respect to Φ.
  - $\Phi(v_i) = \Phi(v_i) - \alpha \frac{dJ}{d\Phi}$

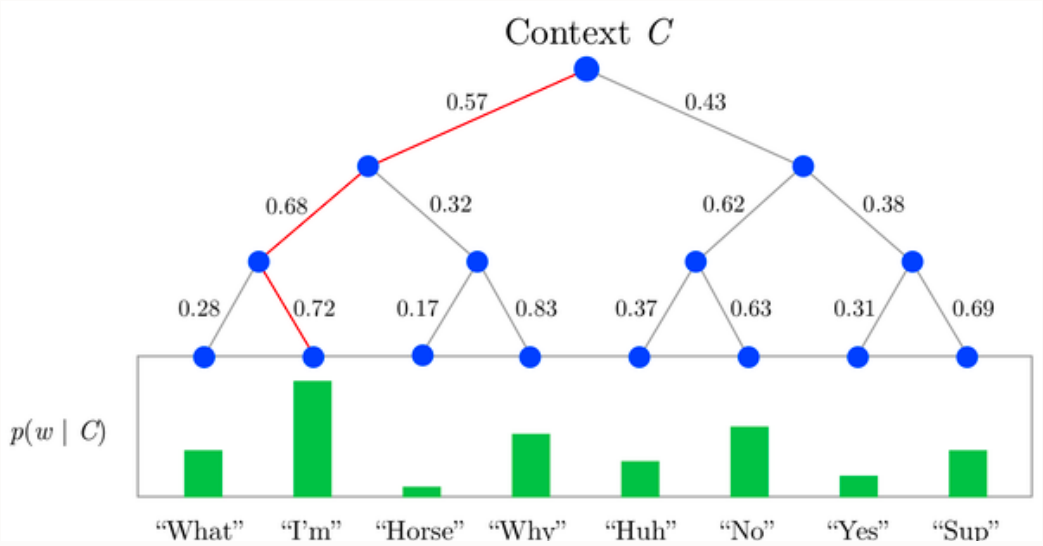
- The loss function is to minimise (with respect to Φ)
  - $J = \min_{\Phi} -\log \Pr(v_{i+l}|\Phi(v_i))$
    - $l \in \{-w, \cdots, -1, 1, \cdots, +w\}$
  - Probability is modelled as $\Pr(v_{i+l}|v_i) = \frac{\exp(v_{i+l}{}^{\intercal}\Phi(v_i))}{\sum_{v \in V} \exp(v^{\intercal}\Phi(v_i))}$

- Important note – also v_{i+l} stands for vector representation of node v_{i+l}, but by using $\Phi(v_i)$ for vector representation of v_i we want to emphasise that we will only update vector representation of v_i.

Hierarchical SoftMax

- The probability $\Pr(v_{i+l}|v_i) = \frac{\exp(v_{i+l}{}^\intercal \Phi(v_i))}{\sum_{v \in V} \exp(v^\intercal \Phi(v_i))}$ is extremely expensive to calculate. The denominator requires to iterate over all vertices in the graph.

- A trick called hierarchical softmax - the idea is to build a binary tree on top of all vertices and edge costs are share of left- and right- subtree.
  - $\Pr(v_{i+l}|\Phi(v_i)) = \prod_{d=1}^{\lceil \log|V| \rceil} \Pr(b_d|\Phi(v_i))$
    - b_d are nodes on path in between root and leaf node.
  - Now the time complexity went from $O(|V|)$ to $O(\log|V|)$
  - Need to build the tree for all random walks around v_i.



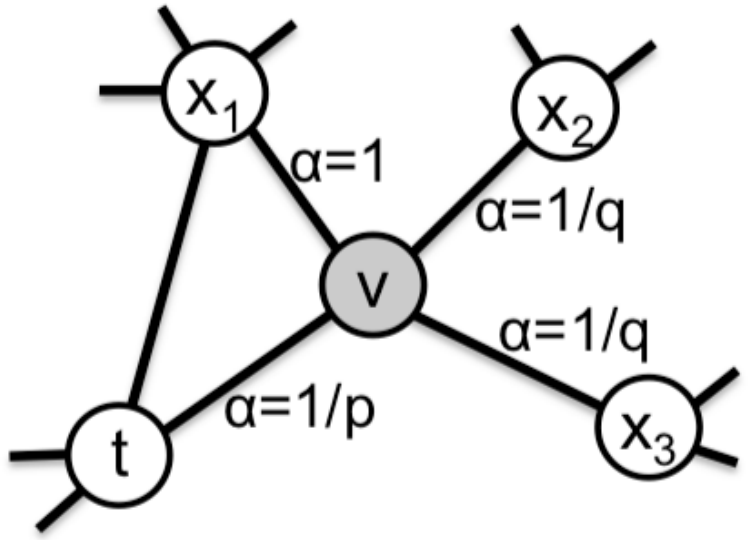
Credit https://www.paperswithcode.com/method/hierarchical-softmax

DeepWalk Wrapup

- DeepWalk builds on NLP model known as Word2Vec.
- Base is a random walk through the graph and cooccurrence of nodes.
- DeepWalk can not handle unseen nodes – the node embeddings are created for nodes present during training.
  - If graph changes, the embeddings have to be recreated.
- In it's basic form, it can not handle values associated with nodes.
- Space Requirements are quite high $O(|V|\times d)$ -- need to store representations of all nodes.

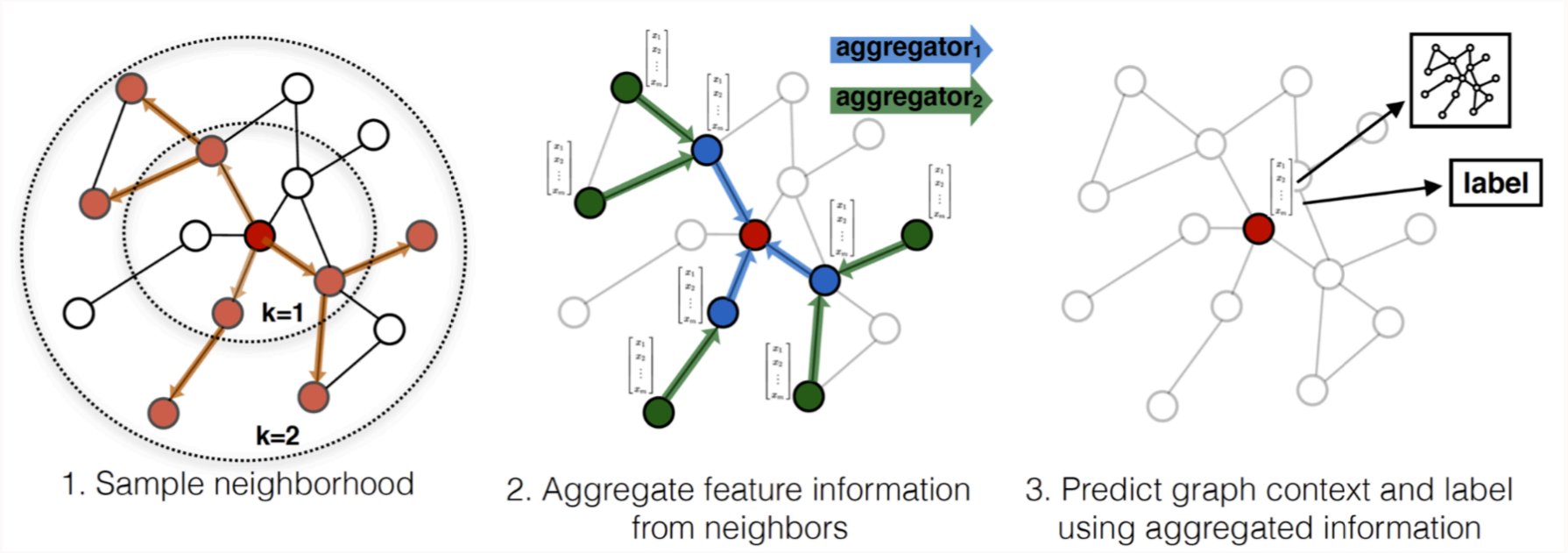
Node2Vec

- Another take on using random walks for vertex embeddings - also inspired by NLP approaches (Word2Vec).
- In contrast to DeepWalk the random walks are not completely random.
  - The choice of next node in the walk is no longer “completely” random.
  - There are parameters controlling exploration on new parts of the graph vs returning to already known parts.
- Transition in previous step was from ***t*** to ***v***. The likelihood for getting from v onwards is influenced by
  - *q* – exploration parameter
  - *p* – return parameter
  - they control how much more/less likely is to return to ***t*** or go further way from it to x_2 or x_3 in comparison to visiting direct neighbour x_1 of ***t***.



GraphSAGE Introduction

- Unlike previous approaches, GraphSAGE (Graph SAmple and aggreGatE) is based on vertex properties and is inductive technique.
  - Does not create embeddings during training phase, but rather trains an neighbourhood aggregator.
  - As consequence, can produce a vector representation even for nodes that were not part of the network during training.



1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

GraphSAGE Algorithm - Inference

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

- x_v are the property values in the node v, they are also initial representation of the node.
- The number of iterations K should be relatively small.

GraphSAGE – Aggregate and Concatenate

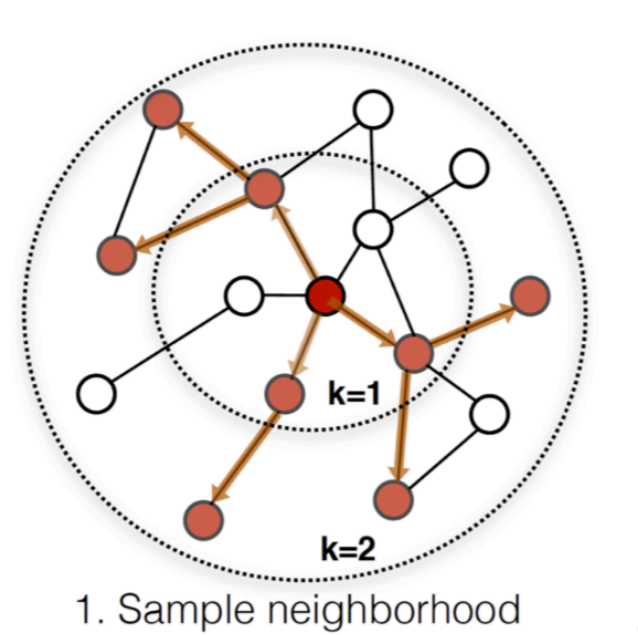
- First, do $AGGREGATE_k$ – combine vertex representation associated with nodes in neighbourhood.
  - MEAN – does aggregation and concatenation in one step by averaging the representations of neighbours and the node itself.
    - $h_v^k \leftarrow \sigma(\mathbb{W}\ \text{MEAN}(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$
  - LSTM – uses LSTM cells to do aggregation. More expressive, but more complicated to train.
  - POOL – similar to pooling from CNN, idea is to represent the neighbourhood by it's strongest signal - $\max\left(\left\{\sigma\left(\mathbb{W}_{pool}\ h_{u_i}^k + b\right), \forall u_i \in \mathcal{N}(v)\right\}\right)$.
    - Max operation is applied elementwise.
- $CONCAT_k$ – combines vertex representation of present node with aggregated neighbourhood.
  - $CONCAT_k$ is a fully connected feed forward network.

$$\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\});$$
$$\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$$

Neighbourhood Sampling and Number of Iterations

- When identifying neighbourhood $N(v)$, only a subset of neighbourhood is taken into consideration.
  - Subset is limited to specified size – reason is to avoid worst-case scenario, where $\|N(v)\| = \|V\|$.

- In each iteration of GraphSAGE, information from a particular node is propagated one step further.



Learning Weight Matrices (1)

- We need to find weight matrix in CONCAT step and train AGGREGATE function, if necessary:
  - in POOL aggregator, the separate weight matrix $\mathbb{W}_{pool}$.
  - in LSTM aggregator, we need to train the LSTM cells.
- Training is unsupervised with negative sampling.
  - Idea that nodes with similar neighbourhood (with similar property values) has same representations, while other nodes has as dissimilar node representation as possible.
- The loss function: $J_{\mathcal{G}}(z_u) = -\log\left(\sigma(z_u \intercal z_v)\right) - Q \times \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma\left(-z_u \intercal z_{v_n}\right)\right)$
  - z_v - is representation of a node at the end of k step random walk.
  - $P_n(v)$ - represent the negative sampling distribution.
  - σ – sigmoid function

$$\mathbf{h}_{\mathcal{N}(v)}^{k} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\});$$
$$\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$$

Learning Weight Matrices (2)

- The training is short one (for ML).
- For training of the weight matrices is performed using (Stochastic) Gradient Descent.
  - The z_v, z_u and z_{v_i} are updated based on present values of the weight matrices.

- Number of iterations is equal to size of influence neighbourhood - K in the algorithm.
  - Each vertex will be presented to the optimisation algorithm at most K-times.
  - The assumption is that the number of gradient steps, due to number of vertices, will be sufficient to perform the gradient descent despite low number of iterations.

GraphSAGE Wrap Up

- Node representation is based on values associated with them.
- Node neighbourhood is implicit, in the values associated with neighbouring nodes.
- Not building vector representations directly, instead building models to generate the representation based on node and neighbourhood values.
- Can infer vector representation for nodes not seen during training.
- Space Requirements are much lower – no need to store vector representation of all nodes.

Graph2Vec

- Build vector representations of whole graphs.

- Similarly to Node2Vec, the Graph2Vec is inspired by NLP approach – specifically Doc2Vec.
  - In Doc2Vec we estimate probability of a word occurring in specific document.

- The document in Graph2Vec case is the graph and words correspond to labels of nodes in the graph.
  - In Doc2Vec the representation is based on words in document, in Graph2Vec the representation is based on subgraphs rooted in nodes of the graph.

- Have a set of graphs $\mathbb{G} = \{G_1, G_2, G_3, \cdots\}$
  - Each graph G_i has it's set on nodes $\mathcal{N}$, edges $\mathcal{E} \subseteq (\mathcal{N} \times \mathcal{N})$ and labels λ: $\mathcal{N} \rightarrow \ell$.
  - sg_i is subgraph of G_i in intuitive meaning.
  - We are looking for graph representation Φ: $\mathbb{G} \rightarrow \mathbb{R}^{\delta}$. Φ is again a lookup table.

Graph2Vec

- Start with random vector representations Φ.
- Run $\mathfrak{e}$ epochs.
- In each epoch, process every graph in random order.
- In the graph visit all nodes and in each node, generate D subgraphs with increasing "diameter" of the subgraph from 0 to D.
- With each subgraph, calculate the loss function and update the embeddings.

input : $\mathbb{G} = \{G_1, G_2, ..., G_n\}$: Set of graphs such that each graph $G_i = (N_i, E_i, \lambda_i)$ for which embeddings have to be learnt
D: Maximum degree of rooted subgraphs to be considered for learning embeddings. This will produce a vocabulary of subgraphs, $SG_{vocab} = \{sg_1, sg_2, ...\}$ from all the graphs in $\mathbb{G}$
δ: number of dimensions (embedding size)
$\mathfrak{e}$: number of epochs
α: Learning rate

output: Matrix of vector representations of graphs $\Phi \in \mathbb{R}^{|\mathbb{G}| \times \delta}$

begin
  Initialization: Sample Φ from $R^{|\mathbb{G}| \times \delta}$
  for $e = 1$ *to* $\mathfrak{e}$ **do**
    $\mathfrak{G} =$ SHUFFLE $(\mathbb{G})$
    for ***each*** $G_i \in \mathfrak{G}$ **do**
      for ***each*** $n \in N_i$ **do**
        for $d = 0$ *to* D **do**
          $sg_n^{(d)} :=$ GETWLSUBGRAPH(n, G_i, d)
          $J(\Phi) = -\log \Pr\ (sg_n^{(d)} | \Phi(G))$
          $\Phi = \Phi - \alpha \frac{\partial J}{\partial \Phi}$
  return Φ

Graph2Vec Loss Function, Negative Sampling and Optimisation

- The loss function to optimise for i-th graph G_i in d-th degree of rooted subgraph:
  - $\max \sum_{j}^{\|\mathbb{G}\|} \log \Pr(sg_n^{(d)} | G_i)$
  - The probability of $\Pr(sg_n^{(d)} | G_i) = \frac{\exp\left(\Phi(G_i) \cdot sg_n^{(d)}\right)}{\sum_{\mathrm{sgn}(d)} \exp\left(\Phi(G_i) \cdot sgn^{(d)}\right)}$
    - Similarly to Node2Vec denominator is extremely expensive to get.
  - Instead of hierarchical soft-max, uses another trick – negative sampling.
    - No need to evaluate all possible subgraphs, use only a small subset.
    - Get a sample of subgraphs from other graphs and use them to normalise the probability.

- For optimisation procedure, use stochastic gradient descent.

Additional Information

- On techniques in the presentation:
  - DeepWalk – https://arxiv.org/pdf/1403.6652.pdf
  - Node2Vec – https://arxiv.org/pdf/1607.00653.pdf
  - GraphSAGE – https://arxiv.org/pdf/1706.02216.pdf
  - Graph2Vec – https://arxiv.org/pdf/1707.05005.pdf

- Additional techniques:
  - Deep Graph Infomax - https://arxiv.org/pdf/1809.10341.pdf
  - A Comprehensive Survey on Graph Neural Networks – https://arxiv.org/pdf/1901.00596.pdf

Libraries to use

- StellarGraph library (on top of Tensorflow)
  - https://github.com/stellargraph/stellargraph
  - List of implemented algorithms: https://stellargraph.readthedocs.io/en/stable/README.html#algorithms
    - Node2Vec: https://stellargraph.readthedocs.io/en/stable/api.html#node2vec
    - GraphSage: https://stellargraph.readthedocs.io/en/stable/api.html#graphsage

- PGL (on top of PyTorch)
  - https://github.com/PaddlePaddle/PGL
  - List of implemented algorithms: https://pgl.readthedocs.io/en/latest/#model-zoo

- Deep Graph Library (on top of PyTorch)
  - https://github.com/dmlc/dgl
  - Contains mostly Graph Convolutional Network and it's variations
    - https://docs.dgl.ai/tutorials/models/index.html

Contest Definition

Contest Goal

- Your task is help an analyst to extend core case graph to final case graph in automatic way.
  - In other words, build a ML model identifying nodes to include the core financial case graph.

We provide a synthetic financial graph with core case graphs identified and some of them (learning cases) have marked nodes that should extend the core case graph.

Vertex and Edge Types

- There are following types of nodes in the graph:
  - Customer – person and organisation
  - Account
  - Address
  - Derived Entity
  - External Entity
- The edges have following types:
  - Money transfer
  - Similar to
  - Has account

Contest Related Case Graph Properties

- The core case graph is a subgraph of financial graph, nodes belonging to a core case graph are marked by same value in **`CoreCaseGraphID`** property.
- The nodes to include in the final case graph are marked by the same value in **`ExtendedCaseGraphID`** property – ground truth information to use for learning.
  - These nodes are to be identified by ML model.
- The nodes, core case graphs, to be used for final performance measurement are marked by testingFlag=1; nodes testingFlag=0 can be used in training and evaluation of the model.
  - Note that nodes in neighbourhood of core case graphs with testing flag set do not have ExtendedCaseGraphID filled.

Generic Graph Properties

- Apart from properties mentioned on previous slide, the graph contains few properties to make graph more realistic.

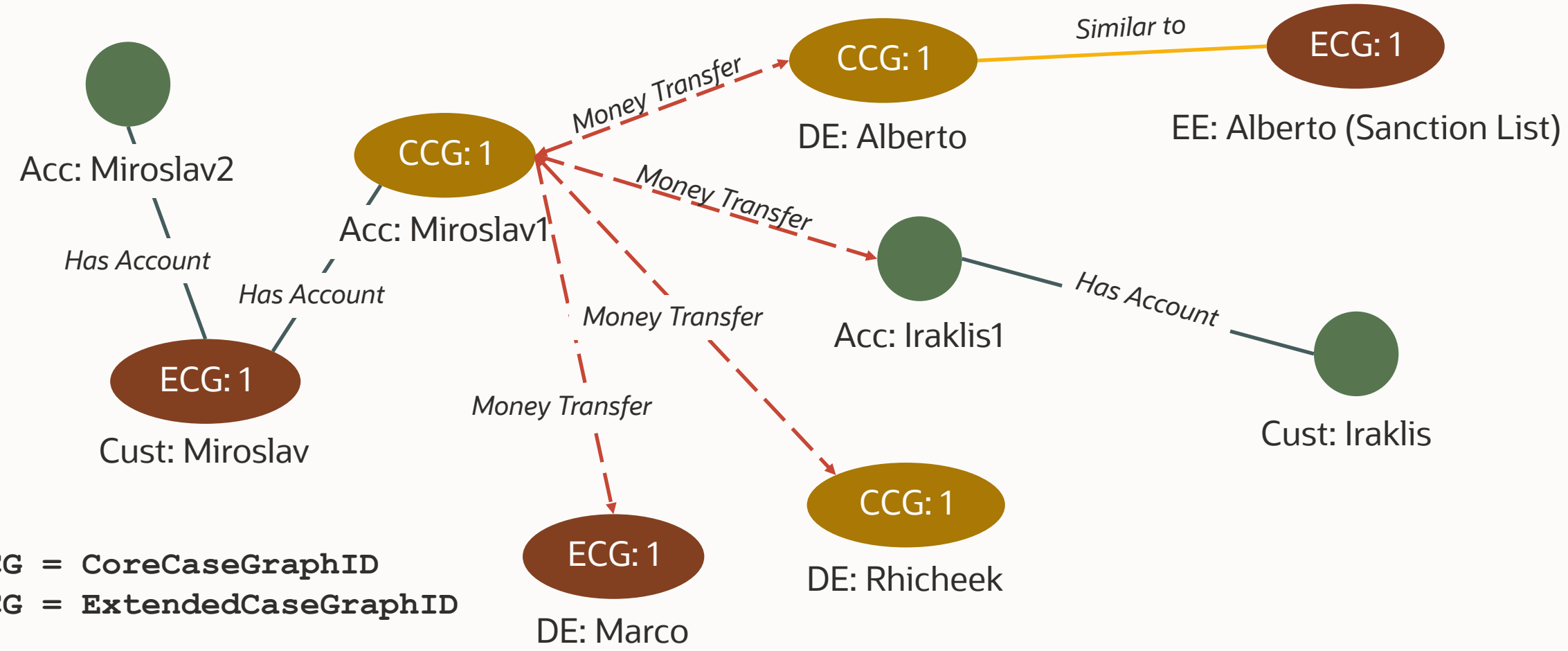
Vertex Properties

Customer	Account	Address	Derived Entity	External Entity
Person / Organisation Flag	High / Mid_High / Medium / Mid_Low / Low Revenue	Address	Person / Organisation Flag	Person / Organisation Flag
Name	ID String		Name	Name
High / Medium / Low Income Flag				

Edge Properties

Money Transfer	Has Account	Has Address	Is similar to
Small / Medium / Large Amount			Strong / Medium / Weak Similarity

Financial Graph Example




```
CCG = CoreCaseGraphID
ECG = ExtendedCaseGraphID
```

Performance Metric

The performance of your algorithm is evaluated using F1 score:

- $F1 = 2\,\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$

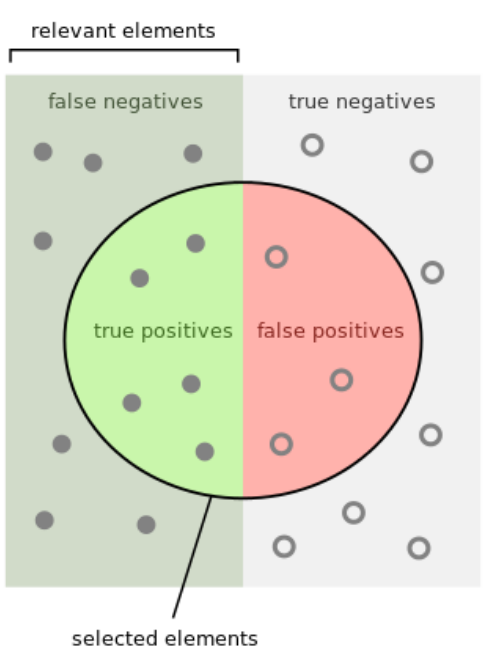
The recall is calculated as follows:

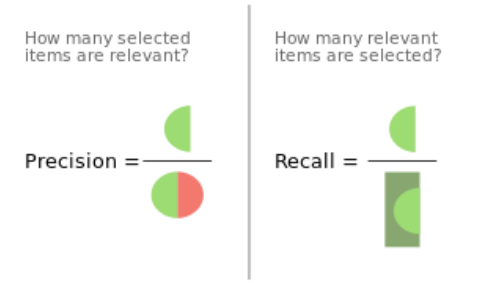
- $\frac{\#\,\text{True Positives}}{\#\text{True Positives} + \#\text{False Negatives}}$

The precision is calculated as follows:

- $\frac{\#\text{True Positives}}{\#\text{True Positive} + \#\text{False Positives}}$

We have prepared a python script you can use to calculate the F1 score of your ML model.





	Predicted Part of Case	Predicted not in Case
Actually Part of a Case	True Positive	False Negative
Actually not in a Case	False Positive	True Negative

Submission

Your submission have to include:

1. A report in PDF describing your approach and model.
2. CSV file containing your predictions for testing cases (the ones with testingFlag=1)
   - The structure of the CSV file is as follows:

NodeID	ExtendedCaseGraphID
N12345	1
N2345	1
N345678	5