metapath2vec: Scalable Representation Learning for Heterogeneous Networks

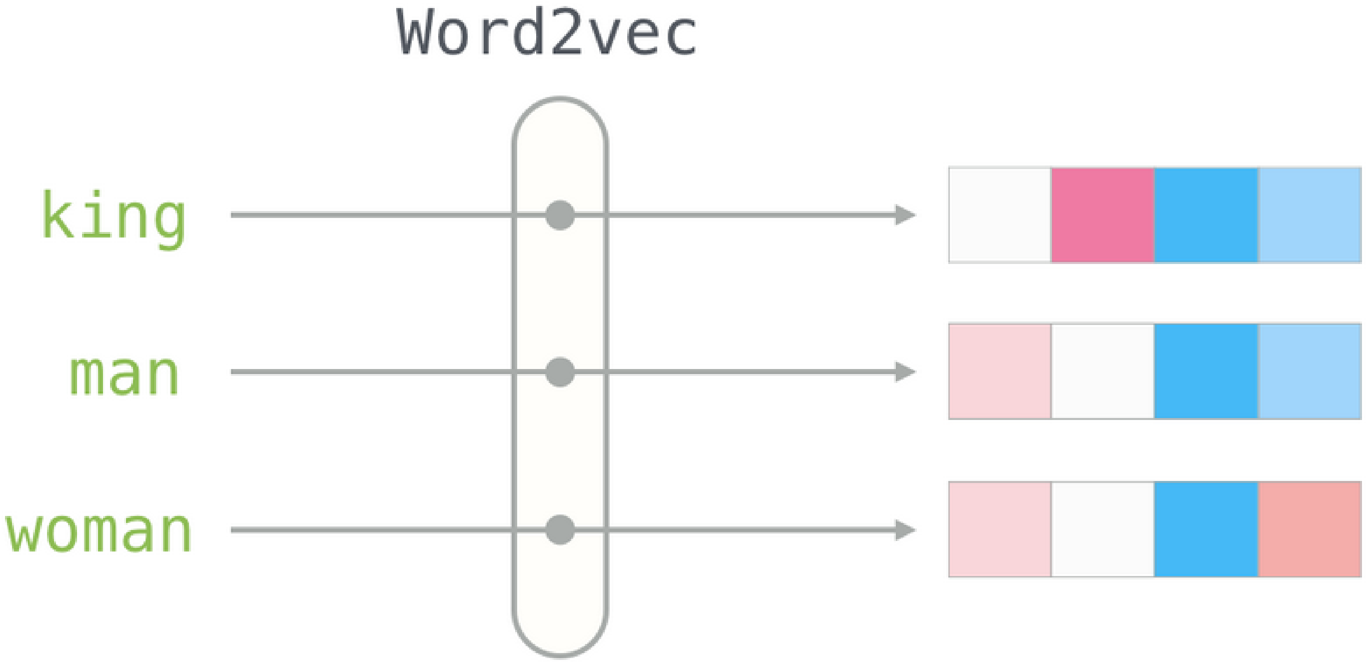
July 25, 2023

김현철

Contents

- Preliminaries
- Paper Review
- Experiments
- Implementation

Preliminaries – (1) Word2Vec



$$[0, 0, 0, 1, 0, 0, 0, 0] \longrightarrow [0.12, 0.49, 0.23, 0.58]$$

Figure from https://jalammar.github.io/illustrated-word2vec/

Preliminaries – (1) Word2Vec

Words in sentences are closely related to their neighbors

Preliminaries – (1) Word2Vec

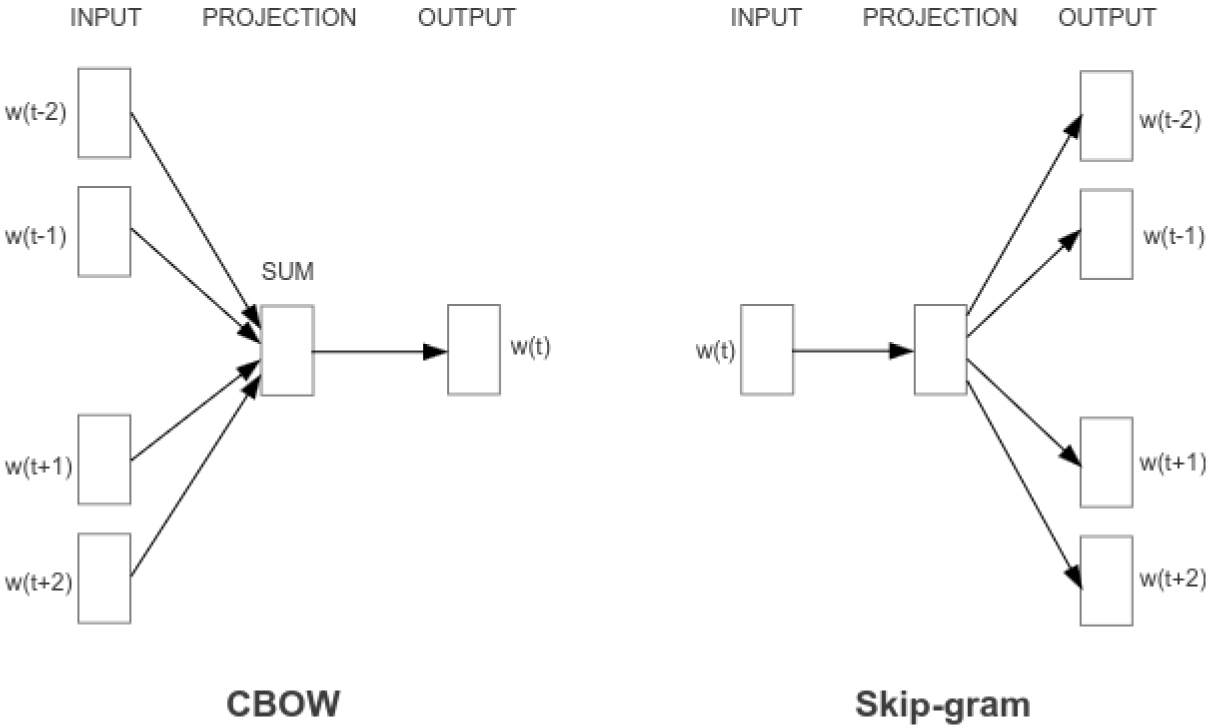


Figure from https://arxiv.org/pdf/1301.3781.pdf

Preliminaries – (1) Word2Vec

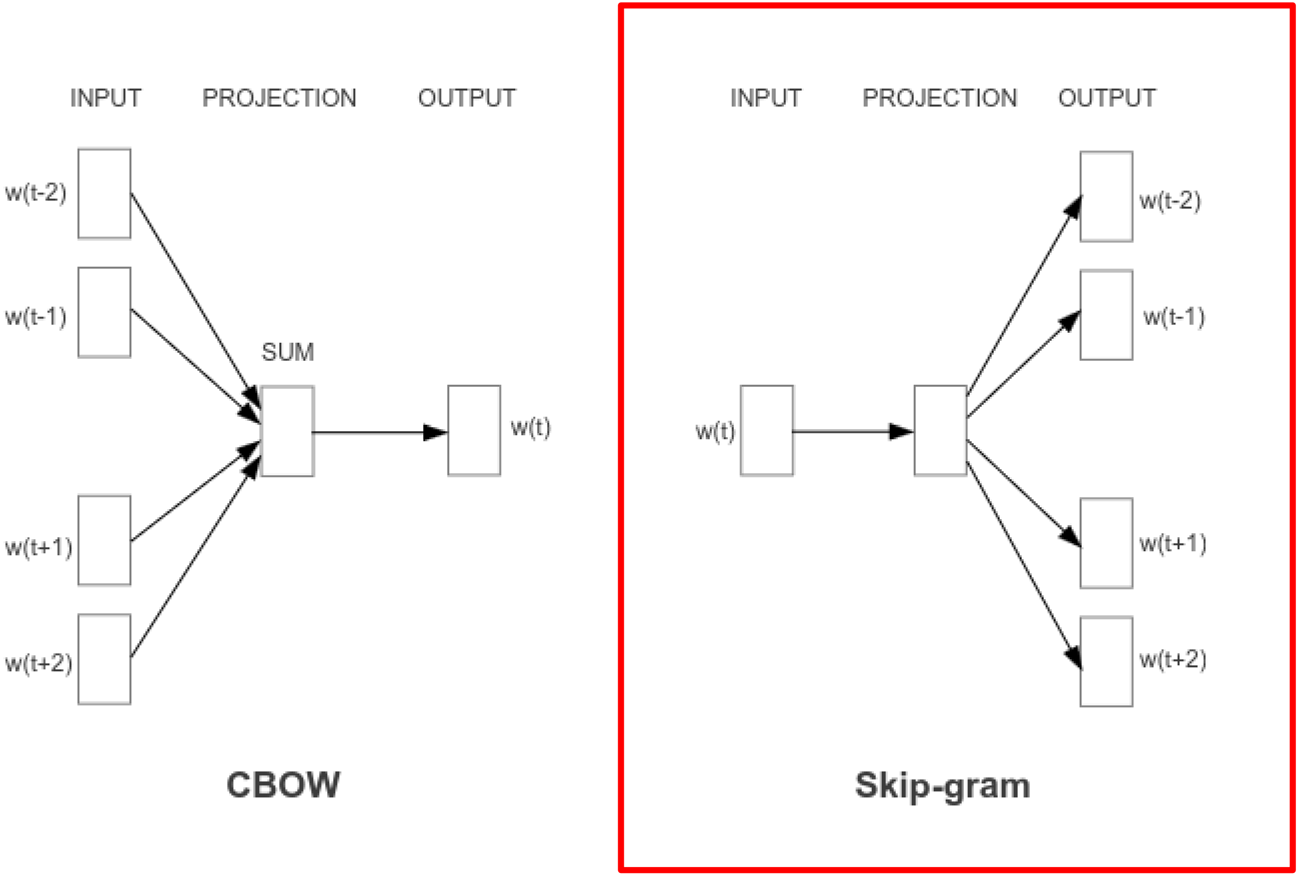
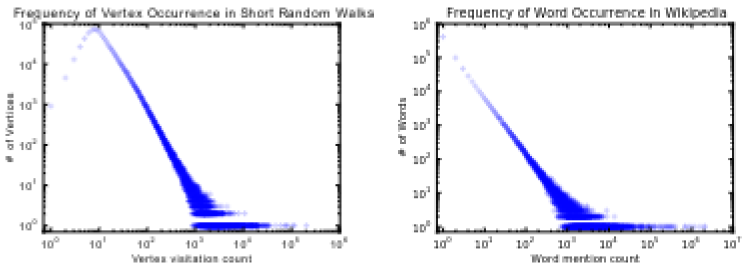


Figure from https://arxiv.org/pdf/1301.3781.pdf

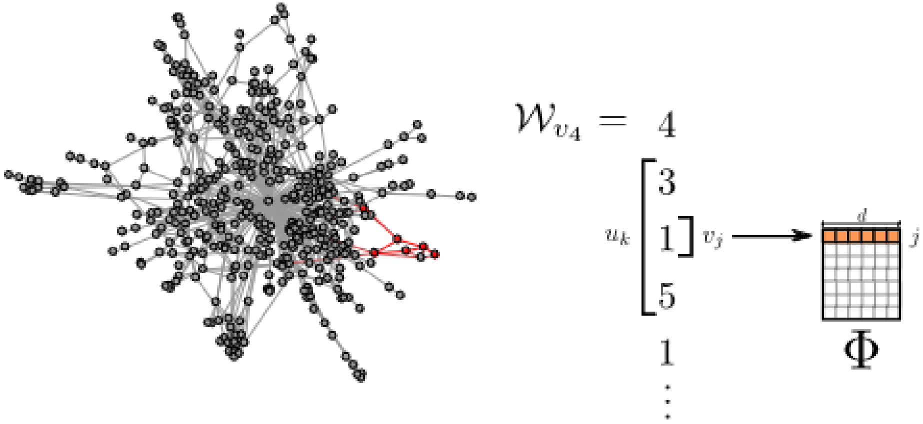
Preliminaries – (2) DeepWalk

: Attempt to embed graph nodes to vectors

- Document(Corpus) → Graph
- Sentence → Random Walk
- Word → Node



(a) YouTube Social Graph (b) Wikipedia Article Text



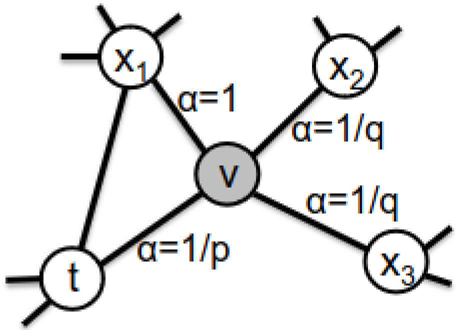
(a) Random walk generation. (b) Representation mapping.

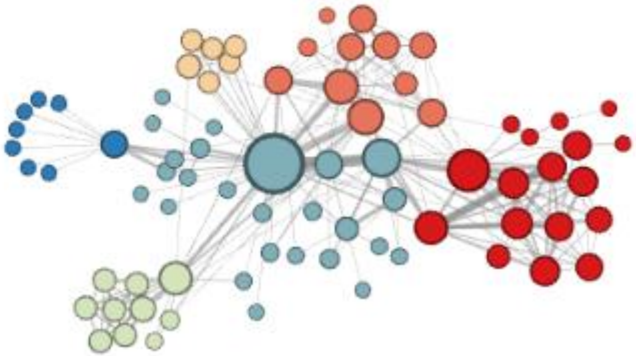
Figures from https://dl.acm.org/doi/pdf/10.1145/2623330.2623732

Preliminaries – (3) Node2Vec

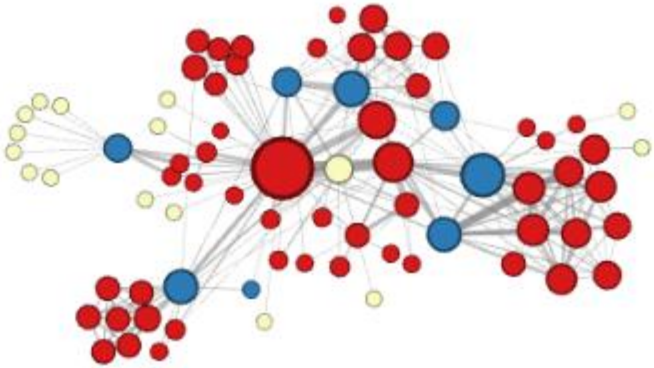
: More general way to generate random walks

- BFS → Learn local feature of the node, i.e, s tructural role
- DFS → Learn global feature of the node, i.e, homophily





DFS



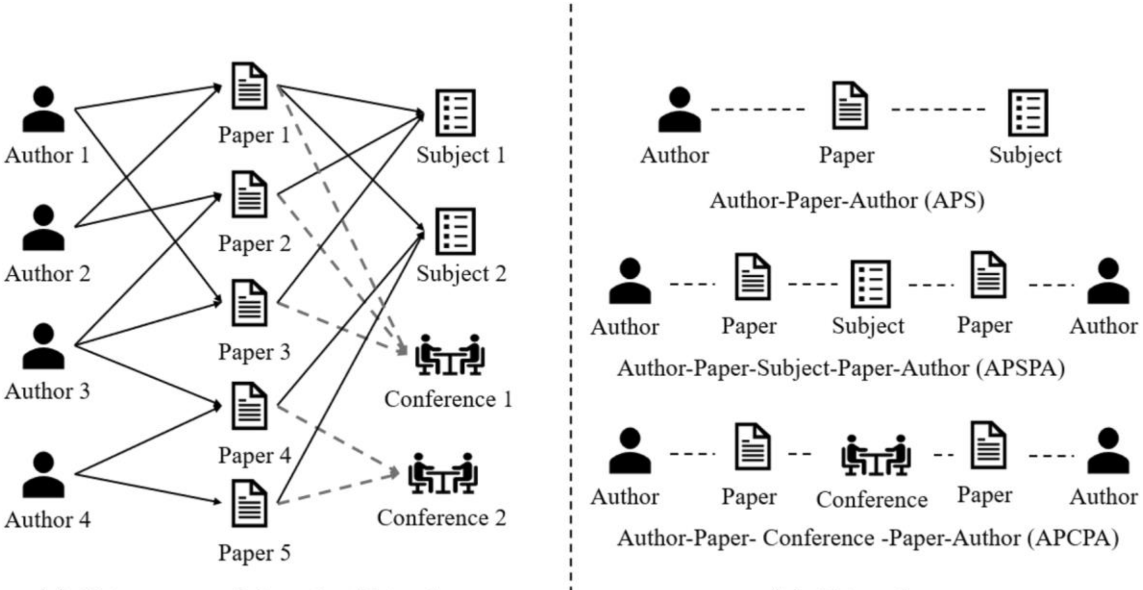
BFS

Figures from https://dl.acm.org/doi/pdf/10.1145/2939672.2939754

Paper Review : Metapath2vec

: Previous works only focus on homogeneous graphs. Thus, node(or edge) types do not affec t random walk process.

However, this is not realistic.



(a) Heterogeneous Information Network (b) Meta-paths

Figures from https://www.nature.com/articles/s41598-023-33324-7

Paper Review : Metapath2vec

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
  π = PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  for $iter = 1$ **to** r **do**
    for all nodes $u \in V$ **do**
      $walk$ = node2vecWalk(G', u, l)
      Append $walk$ to $walks$
  f = StochasticGradientDescent(k, d, $walks$)
  return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
  Inititalize $walk$ to $[u]$
  for $walk\_iter = 1$ **to** l **do**
    $curr = walk[-1]$
    V_{curr} = GetNeighbors($curr, G'$)
    s = AliasSample(V_{curr}, π)
    Append s to $walk$
  return $walk$

Node2vec Algorithm.

Input: The heterogeneous information network $G = (V, E, T)$, a meta-path scheme $\mathcal{P}$, #walks per node w, walk length l, embedding dimension d, neighborhood size k
Output: The latent node embeddings $\mathbf{X} \in \mathbb{R}^{|V| \times d}$

initialize $\mathbf{X}$;
for $i = 1 \rightarrow w$ **do**
  for $v \in V$ **do**
    MP = MetaPathRandomWalk(G, $\mathcal{P}$, v, l) ;
    $\mathbf{X}$ = HeterogeneousSkipGram($\mathbf{X}$, k, MP) ;
  end
end
return $\mathbf{X}$;

MetaPathRandomWalk(G, $\mathcal{P}$, v, l)
$MP[1] = v$;
for $i = 1 \rightarrow l{-}1$ **do**
  draw u according to Eq. 3 ;
  MP[i+1] = u ;
end
return MP ;

HeterogeneousSkipGram($\mathbf{X}$, k, MP)
for $i = 1 \rightarrow l$ **do**
  $v = MP$[i] ;
  for $j = max(0, i\text{-}k) \rightarrow min(i\text{+}k, l)$ & $j \neq i$ **do**
    $c_t = MP$[j] ;
    $X^{new} = X^{old} - \eta \cdot \frac{\partial O(\mathbf{X})}{\partial X}$ (Eq. 7) ;
  end
end

ALGORITHM 1: The *metapath2vec++* Algorithm.

Paper Review : Metapath2vec

Algorithm 1 The *node2vec* algorithm.

LearnFeatures (Graph $G = (V, E, W)$, Dimensions d, Walks per node r, Walk length l, Context size k, Return p, In-out q)
  π = PreprocessModifiedWeights(G, p, q)
  $G' = (V, E, \pi)$
  Initialize $walks$ to Empty
  for $iter = 1$ **to** r **do**
    for all nodes $u \in V$ **do**
      $walk$ = node2vecWalk(G', u, l)
      Append $walk$ to $walks$
  f = StochasticGradientDescent($k, d, walks$)
  return f

node2vecWalk (Graph $G' = (V, E, \pi)$, Start node u, Length l)
  Inititalize $walk$ to $[u]$
  for $walk\_iter = 1$ **to** l **do**
    $curr = walk[-1]$
    V_{curr} = GetNeighbors($curr, G'$)
    s = AliasSample(V_{curr}, π)
    Append s to $walk$
  return $walk$

Node2vec Algorithm.

Regardless of types

Input: The heterogeneous information network $G = (V, E, T)$, a meta-path scheme $\mathcal{P}$, #walks per node w, walk length l, embedding dimension d, neighborhood size k
Output: The latent node embeddings $\mathbf{X} \in \mathbb{R}^{|V| \times d}$

initialize $\mathbf{X}$;
for $i = 1 \to w$ **do**
  for $v \in V$ **do**
    MP = MetaPathRandomWalk($G, \mathcal{P}, v, l$) ;
    $\mathbf{X}$ = HeterogeneousSkipGram($\mathbf{X}, k, MP$) ;
  end
end
return $\mathbf{X}$;

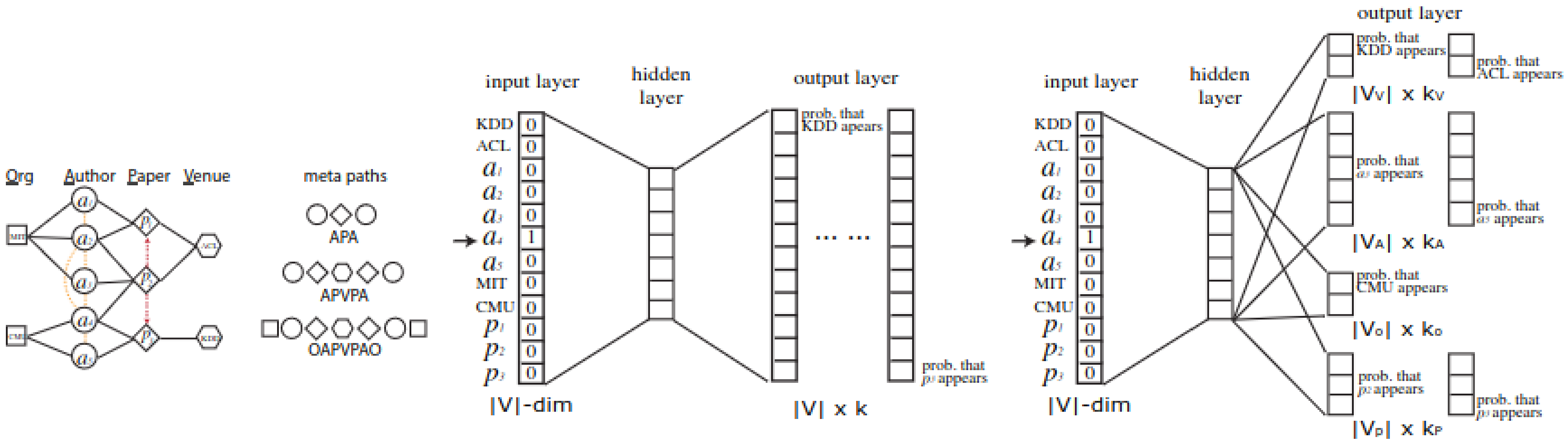
MetaPathRandomWalk($G, \mathcal{P}, v, l$)
$MP[1] = v$;
for $i = 1 \to l{-}1$ **do**
  draw u according to Eq. 3 ;
  $MP[i+1] = u$;
end
return MP ;

HeterogeneousSkipGram($\mathbf{X}, k, MP$)
for $i = 1 \to l$ **do**
  $v = MP[i]$;
  for $j = max(0, i-k) \to min(i+k, l)$ & $j \neq i$ **do**
    $c_t = MP[j]$;
    $X^{new} = X^{old} - \eta \cdot \frac{\partial O(X)}{\partial X}$ (Eq. 7) ;
  end
end

ALGORITHM 1: The *metapath2vec++* Algorithm.

Follows the meta-path

Paper Review : Metapath2vec



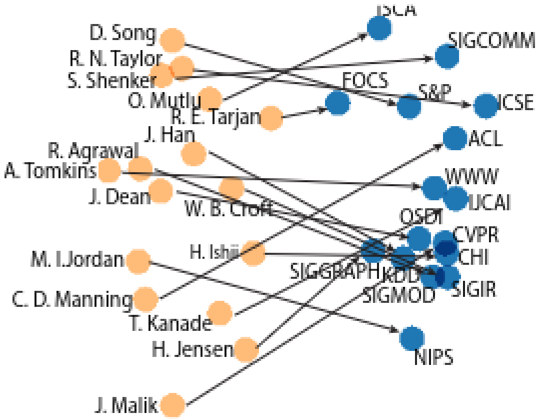
(a) An academic network (b) Skip-gram in *metapath2vec*, node2vec, & DeepWalk (c) Skip-gram in *metapath2vec++*

- Metapath2vec : Compare the probability of every nodes in the graph
  → Maximize the likelihood of preserving both the structure and semantics of a given network.

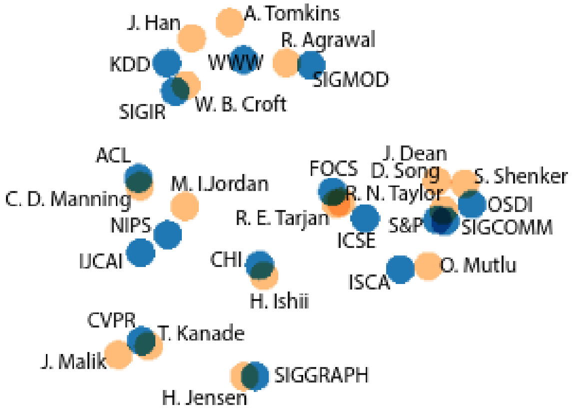
- Metapath2vec++ : Compare the probability of nodes with the same type
  → Accurate and efficient prediction of a node’s heterogeneous neighborhood.

Figures from https://dl.acm.org/doi/pdf/10.1145/3097983.3098036

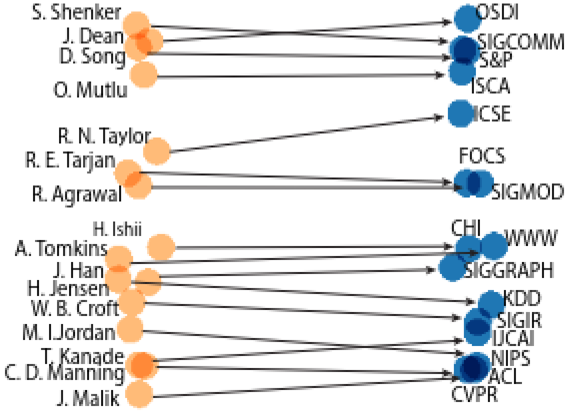
Paper Review : Metapath2vec



(a) DeepWalk / node2vec



(c) *metapath2vec*



(d) *metapath2vec++*

Figures from https://dl.acm.org/doi/pdf/10.1145/3097983.3098036

Experiments

(1) The number of walks per node w: 1000;
(2) The walk length l: 100;
(3) The vector dimension d: 128 (LINE: 128 for each order);
(4) The neighborhood size k: 7;
(5) The size of negative samples: 5.

- Macro-F1 : Average accuracy
- Micro-F1 : Class-wise accuracy

Table 2: Multi-class venue node classification results in AMiner data.

Metric	Method	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Macro-F1	DeepWalk/node2vec	0.0723	0.1396	0.1905	0.2795	0.3427	0.3911	0.4424	0.4774	0.4955	0.4457
	LINE (1st+2nd)	0.2245	0.4629	0.7011	0.8473	0.8953	0.9203	0.9308	0.9466	0.9410	0.9466
	PTE	0.1702	0.3388	0.6535	0.8304	0.8936	0.9210	0.9352	0.9505	0.9525	0.9489
	metapath2vec	0.3033	0.5247	0.8033	0.8971	0.9406	0.9532	0.9529	0.9701	0.9683	0.9670
	metapath2vec++	0.3090	0.5444	0.8049	0.8995	0.9468	0.9580	0.9561	0.9675	0.9533	0.9503
Micro-F1	DeepWalk/node2vec	0.1701	0.2142	0.2486	0.3266	0.3788	0.4090	0.4630	0.4975	0.5259	0.5286
	LINE (1st+2nd)	0.3000	0.5167	0.7159	0.8457	0.8950	0.9209	0.9333	0.9500	0.9556	0.9571
	PTE	0.2512	0.4267	0.6879	0.8372	0.8950	0.9239	0.9352	0.9550	0.9667	0.9571
	metapath2vec	0.4173	0.5975	0.8327	0.9011	0.9400	0.9522	0.9537	0.9725	0.9815	0.9857
	metapath2vec++	0.4331	0.6192	0.8336	0.9032	0.9463	0.9582	0.9574	0.9700	0.9741	0.9786

Table 3: Multi-class author node classification results in AMiner data.

Metric	Method	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%
Macro-F1	DeepWalk/node2vec	0.7153	0.7222	0.7256	0.7270	0.7273	0.7274	0.7273	0.7271	0.7275	0.7275
	LINE (1st+2nd)	0.8849	0.8886	0.8911	0.8921	0.8926	0.8929	0.8934	0.8936	0.8938	0.8934
	PTE	0.8898	0.8940	0.897	0.8982	0.8987	0.8990	0.8997	0.8999	0.9002	0.9005
	metapath2vec	0.9216	0.9262	0.9292	0.9303	0.9309	0.9314	0.9315	0.9316	0.9319	0.9320
	metapath2vec++	0.9107	0.9156	0.9186	0.9199	0.9204	0.9207	0.9207	0.9208	0.9211	0.9212
Micro-F1	DeepWalk/node2vec	0.7312	0.7372	0.7402	0.7414	0.7418	0.7420	0.7419	0.7420	0.7425	0.7425
	LINE (1st+2nd)	0.8936	0.8969	0.8993	0.9002	0.9007	0.9010	0.9015	0.9016	0.9018	0.9017
	PTE	0.8986	0.9023	0.9051	0.9061	0.9066	0.9068	0.9075	0.9077	0.9079	0.9082
	metapath2vec	0.9279	0.9319	0.9346	0.9356	0.9361	0.9365	0.9365	0.9365	0.9367	0.9369
	metapath2vec++	0.9173	0.9217	0.9243	0.9254	0.9259	0.9261	0.9261	0.9262	0.9264	0.9266

Figures from https://dl.acm.org/doi/pdf/10.1145/3097983.3098036

Experiments

Table 5: Case study of similarity search in AMiner Data

Rank	ACL	NIPS	IJCAI	CVPR	FOCS	SOSP	ISCA	S&P	ICSE	SIGGRAPH	SIGCOMM	CHI	KDD	SIGMOD	SIGIR	WWW
0	ACL	NIPS	IJCAI	CVPR	FOCS	SOSP	ISCA	S&P	ICSE	SIGGRAPH	SIGCOMM	CHI	KDD	SIGMOD	SIGIR	WWW
1	EMNLP	ICML	AAAI	ECCV	STOC	TOCS	HPCA	CCS	TOSEM	TOG	CCR	CSCW	SDM	PVLDB	ECIR	WSDM
2	NAACL	AISTATS	AI	ICCV	SICOMP	OSDI	MICRO	NDSS	FSE	SI3D	HotNets	TOCHI	TKDD	ICDE	CIKM	CIKM
3	CL	JMLR	JAIR	IJCV	SODA	HotOS	ASPLOS	USENIX S	ASE	RT	NSDI	UIST	ICDM	DE Bull	IR J	TWEB
4	CoNLL	NC	ECAI	ACCV	A-R	SIGOPS E	PACT	ACSAC	ISSTA	CGF	CoNEXT	DIS	DMKD	VLDBJ	TREC	ICWSM
5	COLING	MLJ	KR	CVIU	TALG	ATC	ICS	JCS	E SE	NPAR	IMC	HCI	KDD E	EDBT	SIGIR F	HT
6	IJCNLP	COLT	AI Mag	BMVC	ICALP	NSDI	HiPEAC	ESORICS	MSR	Vis	TON	MobileHCI	WSDM	TODS	ICTIR	SIGIR
7	NLE	UAI	ICAPS	ICPR	ECCC	OSR	PPOPP	TISS	ESEM	JGT	INFOCOM	INTERACT	CIKM	CIDR	WSDM	KDD
8	ANLP	KDD	CI	EMMCVPR	TOC	ASPLOS	ICCD	ASIACCS	A SE	VisComp	PAM	GROUP	PKDD	SIGMOD R	TOIS	TIT
9	LREC	CVPR	AIPS	T on IP	JAlG	EuroSys	CGO	RAID	ICPC	GI	MobiCom	NordiCHI	ICML	WebDB	IPM	WISE
10	EACL	ECML	UAI	WACV	ITCS	SIGCOMM	ISLPED	CSFW	WICSA	CG	IPTPS	UbiComp	PAKDD	PODS	AIRS	WebSci

Figures from https://dl.acm.org/doi/pdf/10.1145/3097983.3098036

Implementation

- Dataset

  AMiner dataset, reduced to 100,000 authors and 200,000 pa
pers

- Hparams

  Walks per node : 150
  Walk length : 12
  Neighborhood size : 3
  Embedding dimension : 10
  Number of negative samples : 3

  Learning rate : 1e-3
  Batch size : 512

Implementation

```
def calc_prob(AWP, num_author, device):
    prob = torch.zeros(num_author, device=device)

    for i in range(AWP.size(1)):
        prob[AWP[0][i]] += 1

    prob = torch.pow(prob, .75) / math.pow(AWP.size(1), .75)

    return prob
```

Calculates prior distribution for negative sampling

```
def __init__(self, N_author, N_venue, N_paper, prob, args):
    super(metapath2vec, self).__init__()

    self.N_author = N_author
    self.N_venue = N_venue
    self.N_paper = N_paper
    self.N_total = N_author + N_venue + N_paper
    self.prob = prob

    self.path = args.metapath
    self.l = args.walk_len
    self.d = args.d
    self.k = args.neighborhood
    self.lr = args.lr

    # Dim : [author, venue, paper] x d
    self.embedding = nn.Embedding(num_embeddings=self.N_total, embedding_dim=self.d)
```

Initializes embedding matrix

Implementation

```
def walk(self, starting_points, AWP, PPV, VPP, PWA):
    def _random_select(tf):
        ret = torch.zeros(tf.size(0), 1, dtype=int)

        for i in range(tf.size(0)):
            idx = tf[i].nonzero().squeeze(1)
            x = random.randint(0, idx.size(0))
            ret[i] = x

        return ret

    path = starting_points.clone().detach()

    for _ in range(0, self.l, 4): #Only considered APVPA
        p = AWP[1, _random_select(AWP[0, :] == path[:, -1].unsqueeze(1))]
        v = PPV[1, _random_select(PPV[0, :] == p)]
        p = VPP[1, _random_select(VPP[0, :] == v)]
        a = PWA[1, _random_select(PWA[0, :] == p)]

        path = torch.cat([path, a], dim=-1)

    return path
```

Performs random walk

```
def _negative_sample(prob, bound):
    samples = torch.tensor([], dtype=int)

    for i in range(bound.size(0)):
        idx = prob.multinomial(num_samples=self.k).unsqueeze(0)

        while torch.isin(idx, bound[i]).sum().item() != 0:
            idx = prob.multinomial(num_samples=self.k).unsqueeze(0)
        samples = torch.cat([samples, idx], dim=0)

    return samples #[batch_size, N_neg]
```

Performs negative sampling

Implementation

```
optimizer = optim.SGD([self.embedding.weight], lr=self.lr)

for i in range(path.size(1)):
    lbd = max(0, i-self.k)
    rbd = min(path.size(1), i+self.k)

    for j in range(lbd, rbd):
        optimizer.zero_grad()
        pos = torch.log(
                F.sigmoid(
                    (
                    self.embedding.weight[path[:,i]] *
                    self.embedding.weight[path[:,j]]
                    ).sum(dim=1).unsqueeze(1)
                )
            )
```

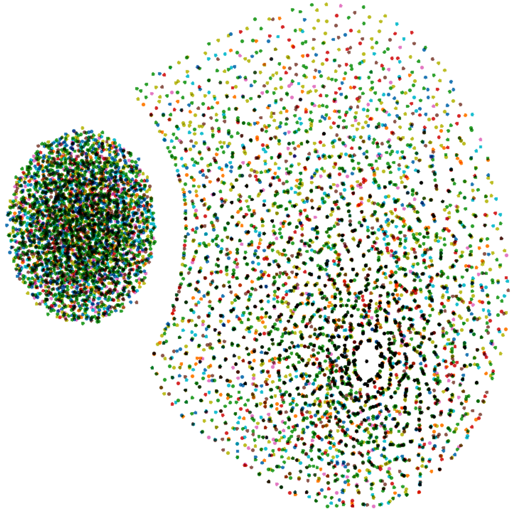
```
neg_samples = _negative_sample(self.prob, path[:, lbd:rbd])
neg = torch.sum(
        torch.log(
            F.sigmoid(
            (
            (-self.embedding.weight[neg_samples].transpose(0,1)) *
            self.embedding.weight[path[:,i]]
            ).sum(dim=2).unsqueeze(2)
            )
        )
        ,dim=0)

O_x = -(pos.mean() + neg.mean())
O_x.backward()
optimizer.step()
```

Performs skip-gram optimization

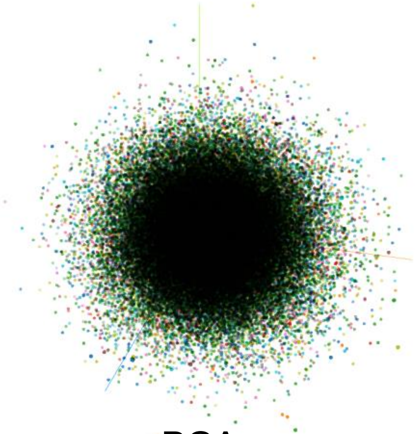
Implementation

```
@torch.no_grad()
def test(args):
    path = os.path.join(args.basedir, args.expname, args.testname)
    model = torch.load(path)


    if args.dataset == 'AMiner':
        dataset = AMiner(root=args.datadir)
    else:
        raise NotImplementedError

    test_embeddings = model['embedding'][dataset[0]['author']['y_index']]
    test_labels = dataset[0]['author']['y']

    writer = SummaryWriter(path)
    writer.add_embedding(
        test_embeddings,
        test_labels
    )

    writer.close()
```

Visualizes embedding space



t-SNE



PCA