表示学习之node2vec

原创 不懂技术的天才 人工智能不要人工 2020-09-15

近年来，知识表示学习在学习自身知识特征进行自动预测领域取得了重大进步。然而，现在的特征学习方法还不足以完全表示网络中已有连接模式的多样性。node2vec为网络节点学习连续的特征表示，通过最大化保留邻接节点的概率，将节点映射到低维的特征空间进行向量表示。

在node2vec之前已经有DeepWalk将word2vec思想运用到图的向量表示。node2vec的主要创新点就是在寻找节点邻域（即节点的上下文信息）的方法做了改进。定义一个灵活的节点的邻域概念和设计有偏的随机游走算法来高效探索邻域的多样性。

node2vec的本质思想来源于word2vec。在给定一个节点，然后通过图搜索算法获取节点的邻域集合，用邻域模拟词的上下文信息。最后使用skip-gram模型训练得到节点的向量。
node2vec将特征表示学习定义为一个最大似然优化问题，定义了一个目标优化函数：

$$\max_f \sum_{u \in V} \log Pr(N_S(u)|f(u)).$$

为了简化优化问题，作了两个假设：
1、条件独立，在给定源节点的特征表示后，观察一个邻域节点的可能性独立于观察任何其他邻域节点。

$$Pr(N_S(u)|f(u)) = \prod_{n_i \in N_S(u)} Pr(n_i|f(u)).$$

2、特征空间对称性，源节点和近邻节点在特征空间中具有对称效应。

$$Pr(n_i|f(u)) = \frac{\exp(f(n_i) \cdot f(u))}{\sum_{v \in V} \exp(f(v) \cdot f(u))}.$$

N_S(u)表示源节点u的邻域(即邻接节点的集合)，n_i表示源节点邻域中的第i个节点，f是将节点映射特征表示的函数，V是网络图的节点集合，v是网络图中除了源节点的其他节点。注：符号"_"表示后面的字符为下标，对应公式中的表示，下同。

node2vec的特征学习是基于skip-gram模型的。而skip-gram模型一开始用来处理自然语言文本的，文本句子是线性的。在给定一段文本，某个词的上下文信息直接使用滑动窗口来获取。但是网络图是非线性的，需要为每个节点定义上下文信息的概念。这里采用的是随机过程采样。

随机过程采样的实现是运用在BFS和DFS的基础上设计有偏的随机游走的方式。假设源节点为c_0=u，模拟固定长度l的随机游走的过程中，第i个节点c_i的概率可以表示为：

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v,x) \in E \\ 0 & \text{otherwise} \end{cases}$$

π_vx为节点v与x之间的未归一化转移概率，Z为归一化常数。

对于有偏变量α的计算，定义带有两个参数p和q的二阶随机游走。假目前一个随机游走走过边(t,v)，目前在节点v出，现在需要考虑下一个节点往哪走，如图所示：

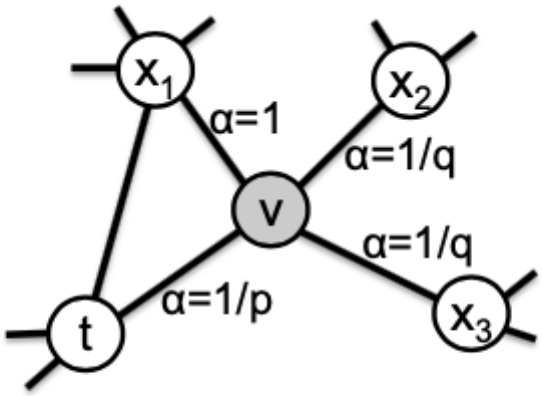

Figure 2: Illustration of the random walk procedure in *node2vec*. The walk just transitioned from t to v and is now evaluating its next step out of node v. Edge labels indicate search biases α.

因此，需要评估随机游走的转移未归一化概率π_vx（即从v游走到下一个节点x的概率）。定义π_vx = α_pq(t,x) · w_vx，其中α_pq(t,x)为边(t,x)的偏置项，w_vx为边的权重，若没有权重则w_vx=1。表达式如图：

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

d_tx表示t到x的最短距离。且取值范围为{0,1,2}。从公式可以看出：

1. 参数p是控制下一步是否需要重新访问上一个节点。如果p值设置的很大（大于max(q,1)），则将基本上不会访问t节点。相反，若p值设置很小（小于min(q,1)），则很大可能重新访问t节点；
2. 参数q是控制下一步游走方向是靠近节点t的节点和是远离节点t的节点。如果q设置为于1，则下一步将访问x1，如果q设置小于1，则下一步将访问x2或x3。也相当于控制随机游走的方式是按广度优先搜索和深度优先搜索的方法进行。

采用随机游走的优点：随机游走在空间上和时间上的计算复杂度都非常高效；

最后，node2vec算法伪代码如下：

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 $walk$

整个过程分三个阶段：边的转移概率的预处理计算、模拟随机游走和使用SGD优化。且每个阶段都可以同步执行。

node2vec源码：

```
import numpy as np
import networkx as nx
import random


class Graph():
  def __init__(self, nx_G, is_directed, p, q):
    self.G = nx_G
    self.is_directed = is_directed
    self.p = p
    self.q = q

  def node2vec_walk(self, walk_length, start_node):
    '''
    Simulate a random walk starting from start node.
    '''
```

```
    G = self.G
    alias_nodes = self.alias_nodes
    alias_edges = self.alias_edges

    walk = [start_node]

    while len(walk) < walk_length:
      cur = walk[-1]
      cur_nbrs = sorted(G.neighbors(cur))
      if len(cur_nbrs) > 0:
        if len(walk) == 1:
          walk.append(cur_nbrs[alias_draw(alias_nodes[cur][0], alias_nodes[c
        else:
          prev = walk[-2]
          next = cur_nbrs[alias_draw(alias_edges[(prev, cur)][0],
            alias_edges[(prev, cur)][1])]
          walk.append(next)
      else:
        break

    return walk

  def simulate_walks(self, num_walks, walk_length):
    '''
    Repeatedly simulate random walks from each node.
    '''
    G = self.G
    walks = []
    nodes = list(G.nodes())
    print 'Walk iteration:'
    for walk_iter in range(num_walks):
      print str(walk_iter+1), '/', str(num_walks)
      random.shuffle(nodes)
      for node in nodes:
        walks.append(self.node2vec_walk(walk_length=walk_length, start_node=

    return walks

  def get_alias_edge(self, src, dst):
    '''
```

```
    Get the alias edge setup lists for a given edge.
    '''
    G = self.G
    p = self.p
    q = self.q

    unnormalized_probs = []
    for dst_nbr in sorted(G.neighbors(dst)):
      if dst_nbr == src:
        unnormalized_probs.append(G[dst][dst_nbr]['weight']/p)
      elif G.has_edge(dst_nbr, src):
        unnormalized_probs.append(G[dst][dst_nbr]['weight'])
      else:
        unnormalized_probs.append(G[dst][dst_nbr]['weight']/q)
    norm_const = sum(unnormalized_probs)
    normalized_probs =  [float(u_prob)/norm_const for u_prob in unnormalized

    return alias_setup(normalized_probs)

  def preprocess_transition_probs(self):
    '''
    Preprocessing of transition probabilities for guiding the random walks.
    '''
    G = self.G
    is_directed = self.is_directed

    alias_nodes = {}
    for node in G.nodes():
      unnormalized_probs = [G[node][nbr]['weight'] for nbr in sorted(G.neigh
      norm_const = sum(unnormalized_probs)
      normalized_probs =  [float(u_prob)/norm_const for u_prob in unnormali
      alias_nodes[node] = alias_setup(normalized_probs)

    alias_edges = {}
    triads = {}

    if is_directed:
      for edge in G.edges():
        alias_edges[edge] = self.get_alias_edge(edge[0], edge[1])
    else:
```

```
        for edge in G.edges():
          alias_edges[edge] = self.get_alias_edge(edge[0], edge[1])
          alias_edges[(edge[1], edge[0])] = self.get_alias_edge(edge[1], edge

      self.alias_nodes = alias_nodes
      self.alias_edges = alias_edges

      return


def alias_setup(probs):
    '''
    Compute utility lists for non-uniform sampling from discrete distributions
    Refer to https://hips.seas.harvard.edu/blog/2013/03/03/the-alias-method-e
    for details
    '''
    K = len(probs)
    q = np.zeros(K)
    J = np.zeros(K, dtype=np.int)

    smaller = []
    larger = []
    for kk, prob in enumerate(probs):
        q[kk] = K*prob
        if q[kk] < 1.0:
            smaller.append(kk)
        else:
            larger.append(kk)

    while len(smaller) > 0 and len(larger) > 0:
        small = smaller.pop()
        large = larger.pop()

        J[small] = large
        q[large] = q[large] + q[small] - 1.0
        if q[large] < 1.0:
            smaller.append(large)
        else:
            larger.append(large)
```

```
    return J, q

def alias_draw(J, q):
    '''
    Draw sample from a non-uniform discrete distribution using alias sampling
    '''
    K = len(J)

    kk = int(np.floor(np.random.rand()*K))
    if np.random.rand() < q[kk]:
        return kk
    else:
        return J[kk]
```

本文的内容主要是阅读node2vec论文《node2vec: Scalable Feature Learning for Networks》的阅读记录。如有错误欢迎留言指正。