图上的机器学习系列-聊聊Node2vec

原创 AaronLou 享受编程的乐趣 2020-04-07

前言

继DeepWalk后，我们再来看一种基于随机游走策略的图嵌入方法——Node2Vec，有点像前者的升级版本，有了前者的基础，理解起来会快很多。

核心方法

Node2Vec与DeepWalk最大的不同（甚至是唯一的不同）就是在于节点序列的生成机制。DeepWalk在每一步探索下一个节点时，是在其邻居节点中进行随机选择，然后基于深度优先策略生成一个固定长度的节点序列。而Node2Vec在生成节点序列时，引入了更加灵活的机制，通过几个超参数来控制向不同方向生长的概率。其核心思路用以下三个图足以充分体现：

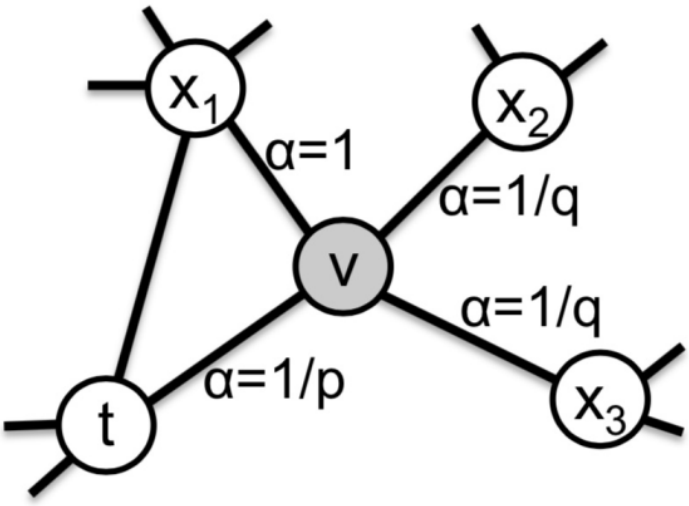


Figure 2: Illustration of the random walk procedure in *node2vec*. The walk just transitioned from t to v and is now evaluating its next step out of node v. Edge labels indicate search biases α.

ized transition probability to $\pi_{vx} = \alpha_{pq}(t,x) \cdot w_{vx}$, where

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

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

在github上可以看其源代码是这样的：

```
def node2vec_walk(self, walk_length, 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[cu
      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
```

可见找到当前节点cur的邻居后，关键就是用alias_draw方法去按某个概率选出来下一个前进的节点。事实上，这个方法并不陌生，在LINE方法的图嵌入（《LINE: Large-scale Information Network Embedding》）当中也使用了同样的技巧。这个方法很有趣，所以可以稍微展开一下。

alias抽样

在讨论方法前，可从代码上感受一下它是干啥的，在Node2vec的源码中可以看到它的实现逻辑很精炼：

```
def alias_setup(probs):
  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]
```

我们手工来一批抽样，感受一下它的产出是怎样的：

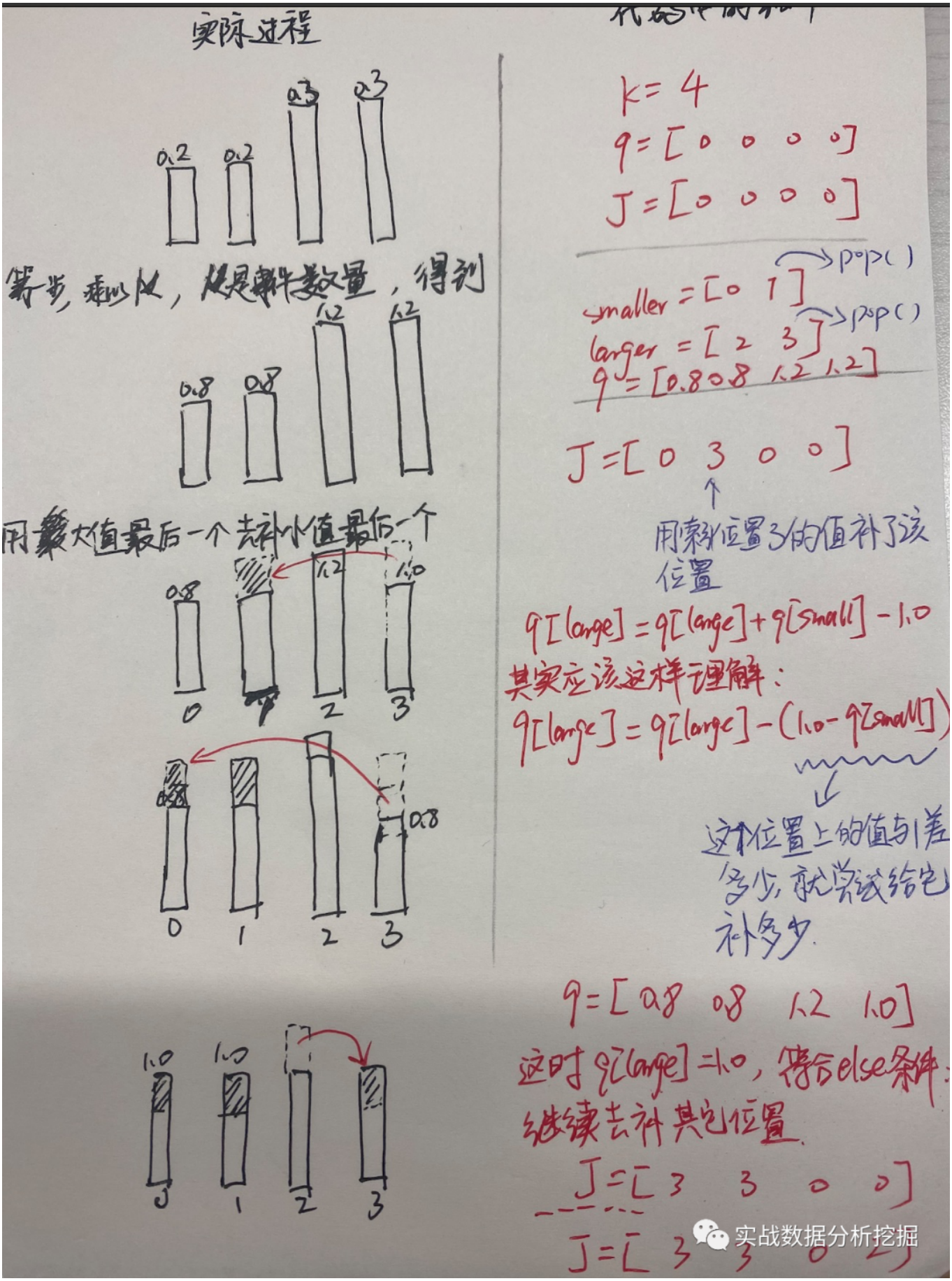
```
from collections import defaultdict
stat = defaultdict(int)
#抽样1000次，统计每个值出现的频率，验证是否符合按指定概率抽样的结果
for i in range(1000):
    # 对于分别以概率0.1，0.2，0.2，0.5出现的事件进行抽样
    J,q  = alias_setup([0.1,0.2,0.2,0.5])
    choice = alias_draw(J,q)  # 得到抽样值
    stat[choice] += 1    # 统计每个值出现的频次

s = sum(stat.values())
for k in range(0,4):
    print(k,stat[k]*1.0/s)  # 计算频率，近似表达概率
```

executed in 0.064s, finished 17:56:58 2020-04-07

```
0 0.097
1 0.197
2 0.204
3 0.502
```

频率统计值与原概率值非常接近，
体现了按概率抽样的结果

可见它实现了一个按指定概率抽样事件的效果，据说这个执行效率是O(1)的，所以应用范围还是较广的。下面来快速了解下内在的执行过程，参考资料中3、4可以用来了解原理。假设我们有事件0，1，2，3，我们想分别以概率0.2, 0.2, 0.3, 0.3来抽样对应的事件，手工示意一下过程中的细节如下图所示：



如果直接在python中执行上述的alias_setup, 可见输出的J数组与示意图中一致，代表了每个位置上被哪个事件来填充过。q数组每个值代表被该位置上数值被其它事件填充前（小于1的时候）分别是多少。

```
alias_setup([0.2,0.2,0.3,0.3])
```

```
executed in 0.124s, finished 18:54:41 2020-04-07
```

```
(array([3, 3, 0, 2]), array([0.8, 0.8, 1. , 0.8]))
```


最后在alias_draw中其实生成了两次随机数字，kk = int(np.floor(np.random.rand()*K)) 生成了一个随机索引值，这一个均匀分布的抽样，抽到每个事件的概率是相等的，都是1/K；然后np.random.rand()又生成了一个(0,1)区间内的随机数，如果这个值小于q数组中对应索引位置上的原始值，则返回该索引位置对应的事件，否则直接返回那个被拿来填充了该位置的事件，而每个位置上被谁填充过，正是已经保存到J数组中了，所以直接读J[kk]即可。

向量化表达

插播结束，继续回来看Node2Vec。根据上述的原则生成了节点序列后，下一步就是进行向量化表达了，这里与DeepWalk就更加统一了，甚至源代码中就是直接引用了gensim.models中的Word2Vec方法。

```
def learn_embeddings(walks):
        '''
        Learn embeddings by optimizing the Skipgram objective using SGD.
        '''
        walks = [map(str, walk) for walk in walks]
        model = Word2Vec(walks, size=args.dimensions, window=args.window_size, min_count=0, sg=1, workers=args.workers, iter=ar
        model.save_word2vec_format(args.output)

        return

def main(args):
        '''
        Pipeline for representational learning for all nodes in a graph.
        '''
        nx_G = read_graph()
        G = node2vec.Graph(nx_G, args.directed, args.p, args.q)
        G.preprocess_transition_probs()
        walks = G.simulate_walks(args.num_walks, args.walk_length)
        learn_embeddings(walks)

if __name__ == "__main__":
        args = parse_args()
        main(args)
```


这个方法执行的过程中使用的一个优化小技巧值得提一提：负采样（Negative Sampling），因为这个方法最近在不同的地方有看到，感觉是个比较有用的思想，所以也可以稍微提一下。

负采样

要解决的问题：每一个训练样本都会去调整SkipGram模型中的每一个参数（这个数量是非常非常多的），严重影响性能。

方法：每一个训练样本仅更新一小部分权重，即一个positive word对应的神经元权重，外加K个negative word对应的神经元权重。每个negative word补选中的概率正比于其词频，一个经验值公式为：

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}(f(w_j)^{3/4})}$$

每个单词被赋予一个权重，即 $f(w_i)$， 它代表着单词出现的频次。

此外

Node2Vec的官方github库中还带有Spark版本的实现 https://github.com/aditya-grover/node2vec/tree/883241e825e1473ef9916ac79f6686f5ef6b1603/node2vec_spark，进一步提升了该方法在我心目中的好感度。

参考资料

1. [http://snap.stanford.edu/node2vec/]
2. [http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/]
3. [https://www.keithschwarz.com/darts-dice-coins/]
4. [https://juejin.im/post/5e71839ce51d452700568d86]