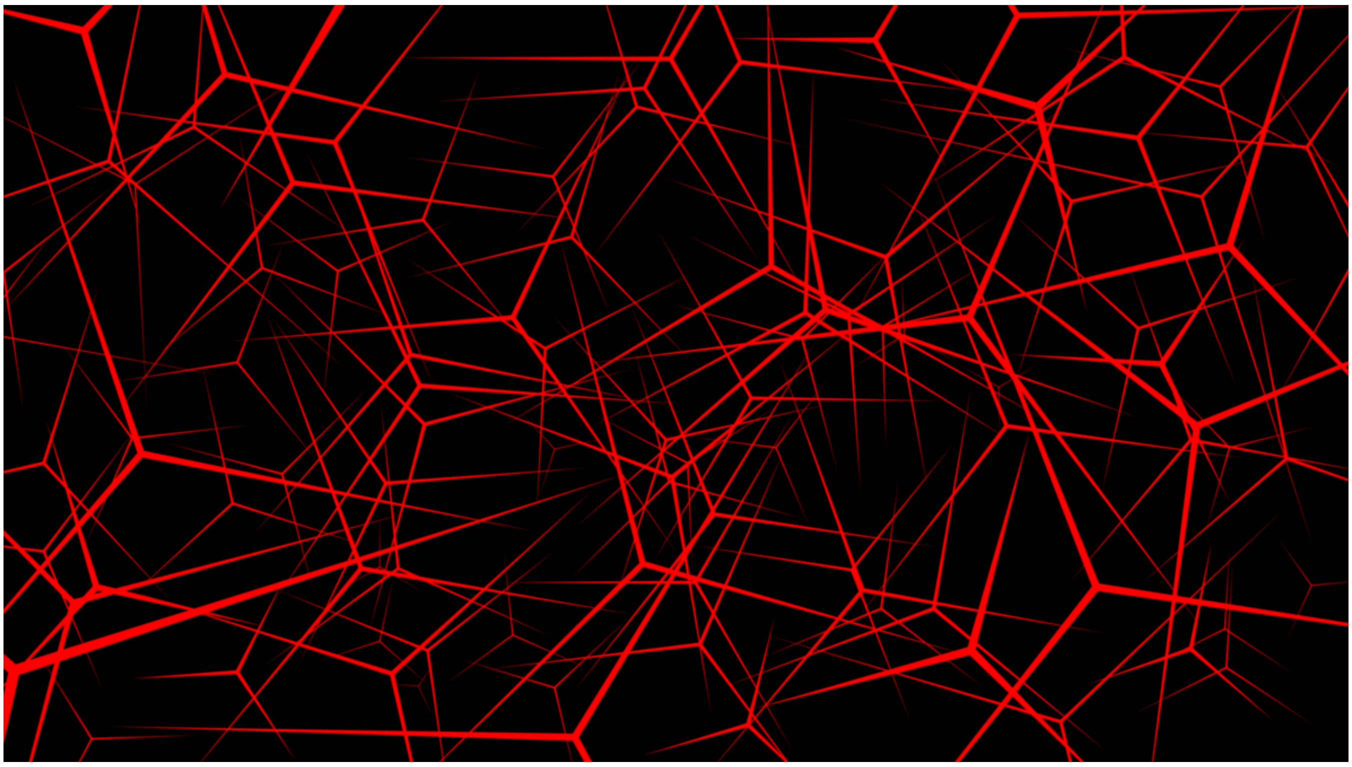
Introduction to Label Propagation with NetworkX — Part 2

Graph ML
Sep 13, 2018 · 4 min read ★



This is Part 2 of the series "Introduction to Propagation with NetworkX". In this post, we will implement Label Propagation using a Python library, NetworkX. If you are not familiar with Label Propagation, you may want to read Part 1 of this series to know the basic concept of the algorithm.

Introduction to Label Propagation with NetworkX — Part 1

Introduction to Label Propagation with NetworkX — Part 2 (this post)

. . .

NetworkX is a Python library by which we can create, manipulate, and analyze the network data. There is no need to "master" it to read this post, but you can try this official tutorial if you would like to. You can install NetworkX using `pip`:

```
pip install networkx
```

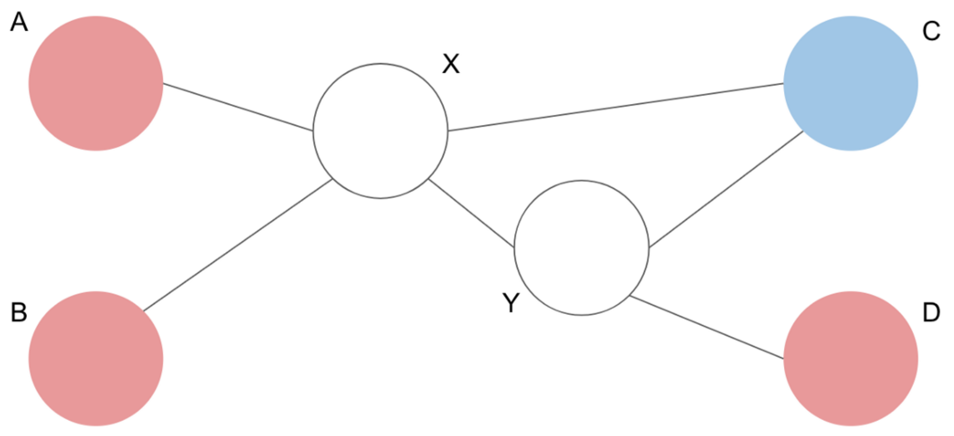
OK, let’s get started. Let’s first create the same toy network as the one that was used in the previous post as an example.

```
import networkx as nx


def create_toy_network():
  G = nx.Graph()
  edges = [('A','X'), ('B','X'), ('X','Y'), ('C','X'), ('C','Y'),
('D','Y')]
  G.add_edges_from(edges)
  G.node['A']['label'] = 'RED'
  G.node['B']['label'] = 'RED'
  G.node['C']['label'] = 'BLUE'
  G.node['D']['label'] = 'RED'
  G.node['X']['label'] = None
  G.node['Y']['label'] = None
  return G


G = create_toy_network()
```

The function `create_toy_network` creates the toy network. It just adds the edges, and assigns labels to the nodes. Let’s illustrate this network again:



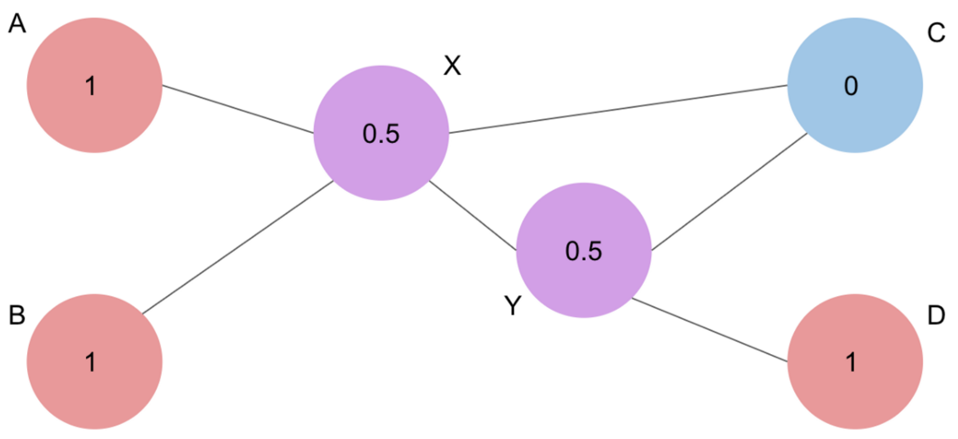
A toy network.

After the toy network is created, let's initialize the "scores" of all nodes. Scores are, as introduced in the previous post, the numbers that Label Propagation calculates to predict the labels. Let's define another function to do that:

```
def initialize_scores(G, init_value):
  for node_id in G.nodes():
    label = G.node[node_id]['label']
    if label == 'RED':
      # Labeled nodes: RED
      G.node[node_id]['score'] = 1
    elif label == 'BLUE':
      # Labeled nodes: BLUE
      G.node[node_id]['score'] = 0
    else:
      # Unlabeled nodes
      G.node[node_id]['score'] = init_value


initialize_scores(G, 0.5)
```

Red nodes are assigned score 0, blue nodes are assigned score 1, and unlabeled nodes are assigned score `init_value` (here it's 0.5). The following figure illustrates the toy network after the nodes are assigned the initial scores:



Initial scores assigned.

The network is ready. It's time to propagate the scores (labels). Recall that the score of an unlabeled node is calculated by taking the average scores of all neighbors. Let's define the function to do that.

```
def calculate_avg_score(G, node_id):
  score_sum = 0
```

```
    n_neighbors = 0
    for neighbor_id in G[node_id]:
      score_sum += G.node[neighbor_id]['score']
      n_neighbors += 1
    return score_sum / n_neighbors


  calculate_avg_score(G, 'X') # => 0.625
  calculate_avg_score(G, 'Y') # => 0.5
```

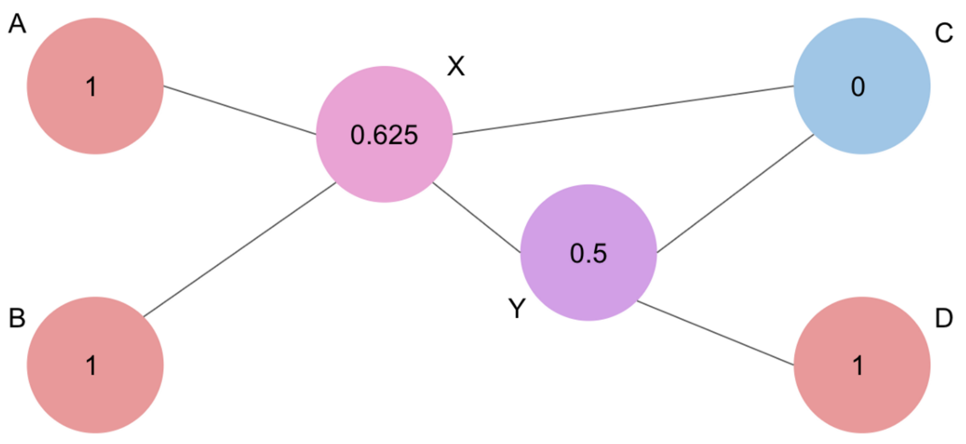
This simple function takes a network and a node ID, then returns the calculated score. As you can calculate by hand, the score of node X is 0.625, and the score of node Y is 0.5, in this case. It's really easy to implement, isn't it?

Finally, let's define function `propagate` that performs one step of propagation:

```
  def propagate(G):
    next_scores = {}
    for node_id in G.nodes():
      if G.node[node_id]['label'] is not None:
        # scores of labeled nodes do not change
        next_scores[node_id] = G.node[node_id]['score']
      else:
        next_scores[node_id] = calculate_avg_score(G, node_id)

    for node_id in next_scores:
      G.node[node_id]['score'] = next_scores[node_id]


  propagate(G) # yields the network shown below.
```



One step of propagate performed.

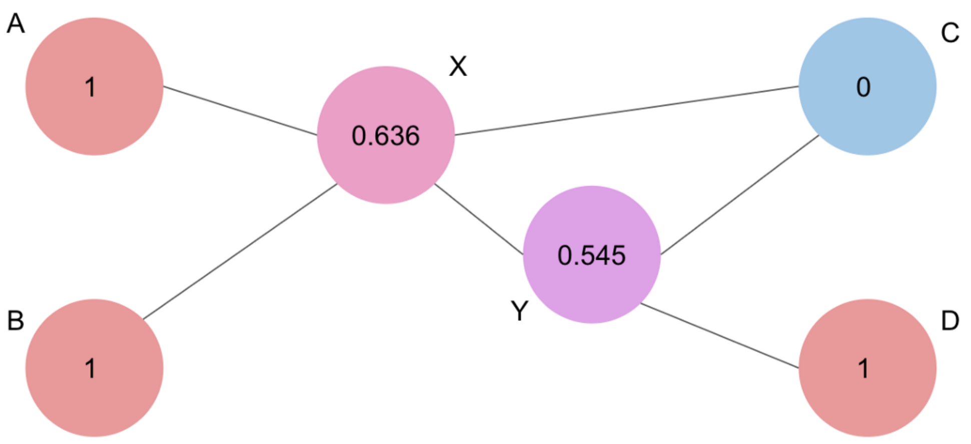
This function just calculates the next scores of all nodes, and update the scores with the calculated next scores. One thing to note is that the scores of all nodes are updated at the same time after all the scores are calculated.

Cool! We have prepared all the utilities to run Label Propagation! Let’s run 10 steps and see what happens:

```
n_steps = 10
for i in range(n_steps):
  propagate(G)
  print("=== After {} steps ===".format(i+1))
  print("X = {}".format(G.node['X']['score']))
  print("Y = {}".format(G.node['Y']['score']))
```

The result shown is:

```
=== After 1 steps ===
X = 0.625
Y = 0.5
=== After 2 steps ===
X = 0.625
Y = 0.541666666667
=== After 3 steps ===
X = 0.635416666667
Y = 0.541666666667
=== After 4 steps ===
X = 0.635416666667
Y = 0.545138888889
=== After 5 steps ===
X = 0.636284722222
Y = 0.545138888889
=== After 6 steps ===
X = 0.636284722222
Y = 0.545428240741
=== After 7 steps ===
X = 0.636357060185
Y = 0.545428240741
=== After 8 steps ===
X = 0.636357060185
Y = 0.545452353395
=== After 9 steps ===
X = 0.636363088349
Y = 0.545452353395
=== After 10 steps ===
X = 0.636363088349
Y = 0.545454362783
```



The network after 10 steps performed.

After 10 steps, the scores mostly converged. And we got the same result as the previous post!

. . .

Congratulations. Now you have an ability to predict the labels of nodes! Although the code we have implemented is only able to handle 2 labels, you now know how it works. If you would like to deal with networks with 3 or more labels, you can use `node_classification` module that will be available from NetworkX version 2.2.

If you enjoyed this series, stay tuned. There are a lot of ML algorithms on network data out there. We'll cover these exciting algorithms one by one!

Machine Learning   Label Propagation   Python   Networkx   Networks