Basline-Model

June 28, 2020

#

Network Science

##

Project #2

1 Description

Consider the flickr dataset (warning, raw data!). File "'*users.txt*" provides a table of form *userID*, *enterTimeStamp*, *additionalInfo...* File "*contacts.txt*" consists of pairs of *userID*'s and link establishment timestamp

Recall *scoring functions* and graph embeddings for link prediction. Your task is to compare the performance of each scoring function as follows: 1. TOP-n accuracy * Denote the number of links E_{new} appeared during testing period as n * Denote the ranked list of node pairs provided by score s as $\hat{E}_s$ * Take top-n pairs from $\hat{E}_s$ and intersect it with E_{new}. Performance is measured as the size of resulted set 2. ROC and AUC ('star' subtask)

Essentially, for this task you also have to follow the guideline points 1 and 2 above. The only thing you have to keep in mind is that flickr dataset is growing dataset. Since then, consider nodes that are significantly represented both in training and testing intervals (have at least 5 adjacent edges in training and testing intervals)

1.1 Solution

Import some useful libraries:

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.xkcd()
import networkx as nx
%matplotlib inline
import pandas as pd

import random
from tqdm.notebook import tqdm
```

1.1.1 Prepare for binary classification problem

[4]:
```python
## download dataset
flickr = pd.read_csv('contacts.txt', header=None, sep=' ')
flickr.columns = ['u1', 'u2', 'time']
## delete negative time data
flickr = flickr[~ (flickr['time'] < 0) ]
## leave nodes with degree >= 5
hist = flickr.stack().value_counts()
sign_nodes = hist[ hist > 4 ].keys()
flickr = flickr[ (flickr['u1'].isin(sign_nodes) & flickr['u2'].
 ↪isin(sign_nodes)) ]

flickr.head(3)
```

[4]:
```
   u1  u2      time
0  41  42  29252304
1  41  44  24407893
2  41  46  28952340
```

Determine datasets with training and testing (gap in the center of time interval) intervals:

[5]:
```python
train_int = flickr[ (flickr['time'] <= np.median(flickr['time'])) ].
 ↪drop_duplicates()
test_int = flickr[ (flickr['time'] > np.median(flickr['time'])) ].
 ↪drop_duplicates()

del train_int['time']
del test_int['time']
```

Make the graph of training interval

[6]:
```python
## make graph of training interval
G_train = nx.from_pandas_edgelist(train_int, 'u1', 'u2',
                                  create_using = nx.Graph())
```

Testing interval should include only authors that have appeared during training interval. Let's find positive examples - we need to find author pairs that haven't been formed during training interval but have been formed by testing interval:

[7]:
```python
## unique users in training and testing intervals and their intersection
unique_train = pd.concat([train_int['u1'], train_int['u2']]).unique()
unique_test = pd.concat([test_int['u1'], test_int['u2']]).unique()
inter = np.intersect1d(unique_test, unique_train)
```

[8]:
```python
## find positive examples
pos = test_int[ (test_int['u1'].isin(inter) & test_int['u2'].isin(inter)) ]
```

Now we should find negative examples such as pairs of users that have not occured together during training and testing intervals. Since it is time-consuming process, we can restrict ourselves with 100,000 of negative examples:

```
[9]: # Create a lookup to speedup the search for negative examples
     # Combine train and test interval
     lookup = dict()
     train_test = train_int.append(test_int)
     for index, user in tqdm(train_test.iterrows(), total= train_test.shape[0]):
         lookup[(user[0], user[1])] = 1
         lookup[(user[1], user[1])] = 1
```

```
HBox(children=(FloatProgress(value=0.0, max=3691084.0), HTML(value='')))
```

```
[10]: """
      ## Find negative examples
      We are randomly selecting two users from the train network and then
      creating connection/pair to check whether that pair exists in the
      lookup(i.e. train and test interval) or not and also shortest distance
      between two users is greater than 3.
      Note:
      Two users are less likely to connect if shorest distance is higher
      between them
      """

      random.seed(0)
      neg_example = set([])
      nodes = list(G_train.nodes)
      pbar = tqdm(total = 100000, desc='Finding Negative Examples')
      count = 0
      while count < 100000:
          user_1 = random.choice(nodes)
          user_2 = random.choice(nodes)
          pair = (user_1, user_2)
          try:
              sp = nx.shortest_path_length(G_train,source=user_1,target=user_2)
          except:
              sp = 100000
          if user_1!=user_2 and sp>3:
              if pair not in neg_example and pair[::-1] not in neg_example:
                  neg_example.add((user_1,user_2))
                  count += 1
                  pbar.update(1)
```

```
HBox(children=(FloatProgress(value=0.0, description='Finding Negative Examples', max=100000.0,
```

[11]:
```
#Convert the set of negative examples into dataframe
neg = pd.DataFrame(neg_example, columns=['u1', 'u2'])
```

[12]:
```
neg.to_csv('neg_example_random_state_0', index=False)
```

So let's make dataset with 100,000 positive and 100,000 negative examples:

[13]:
```
## put your code here

pos_100000 = pos[:100000]
neg_100000 = neg[:100000]

posneg = pos_100000.append(neg_100000).reset_index(drop=True)
```

[14]:
```
posneg.shape
```

[14]:
```
(200000, 2)
```

1.1.2 Construct feature space

So now we got labels for our dataset from training interval. We should build graph of training interval to score some features for pairs of nodes from our labeled dataset. To construct feature space we can take some scoring functions from the lecture (all formulas are provided on slides). This functions reflect relations between each pair of nodes in our dataset and form **baselines**:

- Number of common neighbors
- Preferential attachment
- Jaccard's coefficient
- Adamic/Adar
- Shortest path

[15]:
```
def scoring(df, G):
    common_neighbors = []
    preferential_att = []
    jaccard = []
    adar = []
    shortest_path = []
    for r in df.iterrows():
        nb_v = set(G.neighbors(r[1][0]))
        nb_w = set(G.neighbors(r[1][1]))
        union = nb_v | nb_w
        inter = nb_v & nb_w
        ## Number of common neighbors
        common_neighbors.append(len(inter))
        ## Preferential attachment
        preferential_att.append(len(nb_v) * len(nb_w))
        ## Jaccard's coefficient
        jaccard.append(len(inter) / len(union))
        ## Adamic/Adar
        a = 0
        for node in inter:
            a += 1.0 / np.log(G.degree(node))
        adar.append(a)
        ## Shortest path
        try:
            shortest_path.append(- nx.shortest_path_length(G, source = r[1][0],␣
↪target = r[1][1]))
        except:
            shortest_path.append(- 10000)

    df['common_neighbors'] = pd.Series(common_neighbors, index = df.index)
    df['preferential_att'] = pd.Series(preferential_att, index = df.index)
    df['jaccard'] = pd.Series(jaccard, index = df.index)
    df['adar'] = pd.Series(adar, index = df.index)
    df['shortest_path'] = pd.Series(shortest_path, index = df.index)
```

[16]:
```
scoring(posneg, G_train)
```

1.1.3 TOP-n accuracy

So now we should do the following: for each scoring function let's take top-100000 pairs and intersect it with positive examples:

[17]:
```
submission = pd.DataFrame(columns=['metric','Prediction'])
```

[18]:
```
for score in ['common_neighbors',
              'preferential_att',
              'jaccard',
              'adar',
              'shortest_path']:
    top_n = posneg.sort_values(score, ascending = False)[:100000]
    acc = len(pd.merge(pos_100000, top_n, how = 'inner', on = ['u1', 'u2'])) /␣
↪100000
    print('{}: {}'.format(score, acc))
    submission = submission.append({'metric':score + ' top-100000',
                                    'Prediction':round(acc, 10)},
                                    ignore_index=True)
```

```
common_neighbors: 0.72649
preferential_att: 0.83732
jaccard: 0.72649
adar: 0.72649
shortest_path: 0.77138
```

This rate represents perfomance of scoring functions: we see that preferential attachment gave us best prediction result, bur in general all scoring functions give quite high and similar result - about 70-80%.

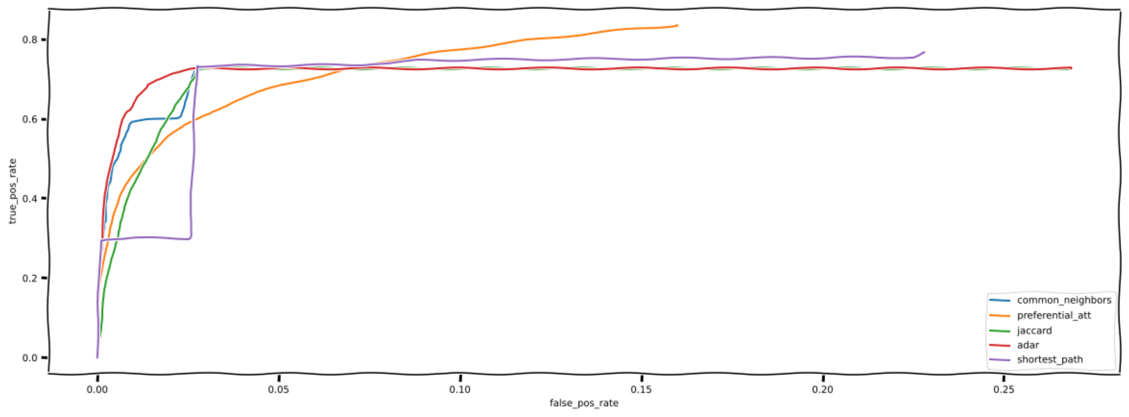
1.1.4 ROC and AUC

Let's also compute ROC and AUC:

[19]:
```
scores = [ 'common_neighbors',
                'preferential_att',
               'jaccard',
                'adar',
                'shortest_path']

plt.figure(figsize=(20,7))

for score in scores:
    true_pos_rate = []
    false_pos_rate = []
    auc = 0
    sort_posneg = posneg.sort_values(score, ascending = False)
    for n in range(0, 100000, 500):
        top_n = sort_posneg[:n]

        tpr = len(pd.merge(pos_100000, top_n, how = 'inner', on = ['u1',␣
 ↪'u2'])) / 100000
        true_pos_rate.append(tpr)
        fpr = len(pd.merge(neg_100000, top_n, how = 'inner', on = ['u1',␣
 ↪'u2'])) / 100000
        false_pos_rate.append(fpr)
    for i in range(1, len(false_pos_rate)):
        auc += (false_pos_rate[i] - false_pos_rate[i-1]) * (true_pos_rate[i-1] +
                                                            true_pos_rate[i]) /␣
 ↪2
    plt.plot(false_pos_rate, true_pos_rate)
    print('For {} AUC = {}'.format(score, auc))
    submission = submission.append({'metric':score + ' AUC',
                                    'Prediction':round(auc, 10)},
                                    ignore_index=True)

plt.xlabel('false_pos_rate')
plt.ylabel('true_pos_rate')
plt.legend(scores, loc = 'best')
```

```
For common_neighbors AUC = 0.19041176495000017
For preferential_att AUC = 0.11242066655000002
For jaccard AUC = 0.18800546495000017
For adar AUC = 0.19225451495000018
For shortest_path AUC = 0.1582429692
```

```
[19]: <matplotlib.legend.Legend at 0x7f2956acbe50>
```



ROC illustrates the performance of a binary classifier system as its discrimination threshold is varied. The curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The ROC curve is the sensitivity as a function of fall-out. AUC is computed as an area under the curve and is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. As we see AUC i rather small for all scoring function (fit our top_n results).

```
[20]: submission.to_csv('submission.csv', index=False)
```

```
[21]: submission
```

```
[21]:                        metric  Prediction
      0  common_neighbors top-100000    0.726490
      1  preferential_att top-100000    0.837320
      2           jaccard top-100000    0.726490
      3              adar top-100000    0.726490
      4     shortest_path top-100000    0.771380
      5         common_neighbors AUC    0.190412
      6         preferential_att AUC    0.112421
      7                  jaccard AUC    0.188005
      8                     adar AUC    0.192255
      9            shortest_path AUC    0.158243
```

```
[ ]:
```