deliverable_3_team_3

January 31, 2019

1 DELIVERABLE 3 - TEAM 3

1.1 Clastres, Baverez, Monoci Neto et Al.

1.2 let's first import twitter data

```
In [178]: import pandas as pd
          import networkx as nx
          import matplotlib.pyplot as plt
          dfs_t = pd.read_excel("twitter_dataset.xlsx", sheet_name= None, date_parser  = int)
          t_post, t_user = pd.read_excel("twitter_posts.xlsx", sheet_name= None), pd.read_exce
```

```
In [179]: import numpy as np
```

Graph Twitter

```
In [57]: G = nx.DiGraph()
         elist = []
         for row in t_user.iterrows():
             row = list(row)
             node, neighbors = int(row[-1][0]),row[-1]['id_followers'][1:-1]
             neighbors = neighbors.split(', ')
             G.add_node(node)
             for n in neighbors:
                 n= int(n)
                 G.add_node(n)
                 elist.append((node,n))
         G.add_edges_from(elist)
```

```
In [58]: len(G.nodes())
```

```
Out[58]: 4098
```

```
In [59]: list(G.nodes())[:3]
```

```
Out[59]: [5662813, 2152321, 3954946]
```

```
In [188]: ## Tracking "fathers" on Twitter
          t_post['infected_by'] = None
```

```
for index, row in t_post.iterrows():
    id_tweet_origin = row['id_tweet_origin']
    if id_tweet_origin != 0:
        infected_by = int(t_post[t_post['id_tweet'] == id_tweet_origin]['id_user'])
        t_post.loc[index,'infected_by'] = infected_by
t_post['id_post'] = t_post['id_tweet']
columns = ['id_user','id_post', 'infected_by', 'date','time','half_day' ]
t_lite = t_post[columns]
t_lite['time']=t_lite["time"].astype('|S')
t_lite = t_lite.sort_values(by=['date', 'half_day','time'])
```

```
/Users/EC/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:11: SettingWithCopyWarnin
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html
  # This is added back by InteractiveShellApp.init_path()
```

Twitter log, sorted chronologically and indexed on id_user

```
In [ ]: t_lite.head(7)
```

1.3 Computation of "required contacts"

```
In [324]: vus = {}
          vect = []
          k=0
          for row in t_lite.iterrows():
              id_user = row[1][0]
              k+=1
              vect.append(len([p for p in G.predecessors(id_user) if p in vus ]))
              vus[id_user]=True
          vect = np.asarray(vect)
          t_lite['required_contacts'] = vect
```

```
In [191]: t_user.head()
```

```
Out[191]:    id_user  nb_followers  nb_following  nb_tweets     sex birth_date  \
          0  5662813            80           431         66  female 1985-11-11
          1  6187946           189           171         75    male 1995-10-01
          2  3122461           175            86        147    male 1982-03-20
          3  1855435           217            35        121  female 1992-03-10
          4  6561414            67           130         37  female 1991-12-03

                                                   id_followers
          0  [2152321, 3954946, 3850627, 9664645, 9576070, ...
          1  [9904643, 7723529, 3763722, 8360535, 5897229, ...
          2  [2096133, 6092033, 8077327, 1885713, 7434770, ...
```

```
3  [6206474, 4367884, 2206737, 3898900, 4445358, ...
4  [1698176, 6092033, 5123435, 7344261, 4777606, ...
```

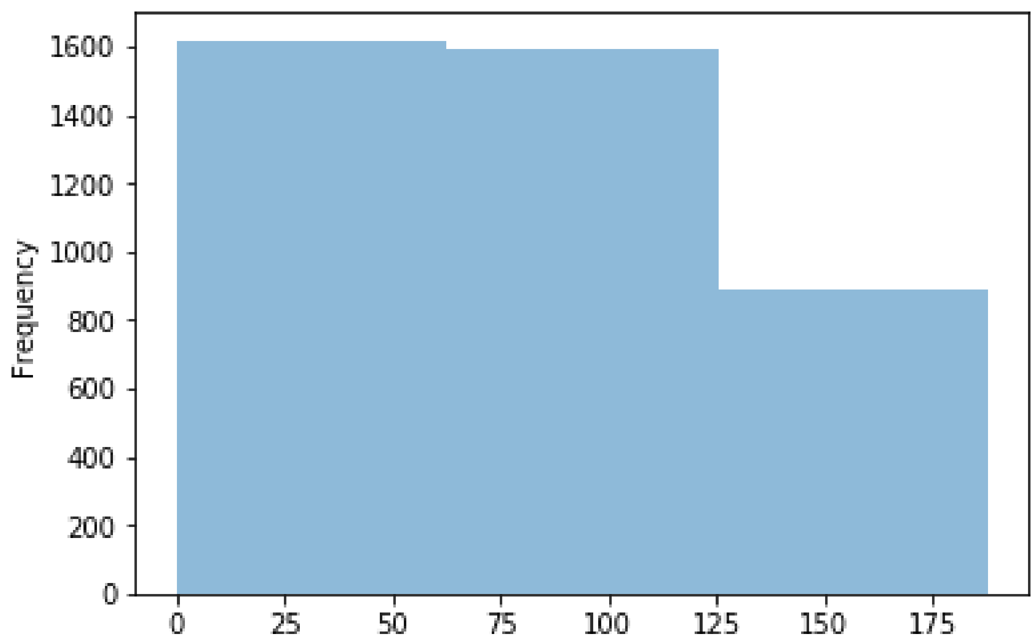
```
In [325]: t_heavy = t_lite.join(t_user.set_index('id_user'), on='id_user')

In [326]: t_heavy.head()

Out[326]:    id_user   id_post infected_by       date         time half_day  \
          5  3003097  48168379        None 2017-11-09  b'02:53:00'       am
          2  6013435  81242015        None 2017-11-09  b'05:18:00'       am
          4  6027974  94580818        None 2017-11-09  b'08:10:00'       am
          1  1953787  55199327        None 2017-11-09  b'08:17:00'       am
          0  9834565  18271010        None 2017-11-09  b'09:41:00'       am

             required_contacts  nb_followers  nb_following  nb_tweets     sex  \
          5                  0           164           106         64  female
          2                  0           233           398         62    male
          4                  1           107           135        133  female
          1                  0           117           100        116  female
          0                  0           211           110         60  female

            birth_date                                        id_followers
          5 1997-07-17  [6749190, 7017991, 1860104, 6980105, 5054990, ...
          2 1990-09-10  [9977349, 1860104, 4184586, 7605591, 8258061, ...
          4 2000-01-05  [3122693, 3578892, 4088853, 9632279, 2002975, ...
          1 2000-12-07  [6084102, 6242827, 5897229, 9359374, 5506067, ...
          0 1986-06-10  [8546819, 5053957, 1863691, 5497357, 8281618, ...

In [327]: t_heavy = t_heavy[t_heavy.columns[:-1]]
          set(t_heavy['sex'].apply(lambda x: x in ['male' , 'female'])) == {True} #y'a t-il de

Out[327]: True

In [328]: t_heavy['sex'] = t_heavy['sex'].apply(lambda x: x == 'male')

In [329]: t_heavy.head()

Out[329]:    id_user   id_post infected_by       date         time half_day  \
          5  3003097  48168379        None 2017-11-09  b'02:53:00'       am
          2  6013435  81242015        None 2017-11-09  b'05:18:00'       am
          4  6027974  94580818        None 2017-11-09  b'08:10:00'       am
          1  1953787  55199327        None 2017-11-09  b'08:17:00'       am
          0  9834565  18271010        None 2017-11-09  b'09:41:00'       am

             required_contacts  nb_followers  nb_following  nb_tweets    sex birth_date
          5                  0           164           106         64  False 1997-07-17
          2                  0           233           398         62   True 1990-09-10
          4                  1           107           135        133  False 2000-01-05
          1                  0           117           100        116  False 2000-12-07
          0                  0           211           110         60  False 1986-06-10
```

```
In [330]: from datetime import date

          def age(date1, date2):
              naive_yrs = date2.year - date1.year
              return naive_yrs

          t_heavy['age'] = t_heavy['birth_date'].map(lambda x: age(x, date.today()))

In [331]: t_heavy.head()

Out[331]:    id_user   id_post infected_by        date         time half_day  \
          5  3003097  48168379        None  2017-11-09  b'02:53:00'       am
          2  6013435  81242015        None  2017-11-09  b'05:18:00'       am
          4  6027974  94580818        None  2017-11-09  b'08:10:00'       am
          1  1953787  55199327        None  2017-11-09  b'08:17:00'       am
          0  9834565  18271010        None  2017-11-09  b'09:41:00'       am

             required_contacts  nb_followers  nb_following  nb_tweets    sex birth_date  \
          5                  0           164           106         64  False 1997-07-17
          2                  0           233           398         62   True 1990-09-10
          4                  1           107           135        133  False 2000-01-05
          1                  0           117           100        116  False 2000-12-07
          0                  0           211           110         60  False 1986-06-10

             age
          5   22
          2   29
          4   19
          1   19
          0   33

In [332]: t_heavy = t_heavy.drop(['birth_date'], axis = 1)

In [333]: #the difference between followers and following may indicate the nature of the user
          t_heavy['diff'] = t_heavy['nb_followers'] - t_heavy['nb_following']

In [334]: t_heavy.describe()

Out[334]:            id_user       id_post  required_contacts  nb_followers  \
          count  4.098000e+03  4.098000e+03        4098.000000   4098.000000
          mean   5.522421e+06  5.522643e+07          80.125183    160.827233
          std    2.548373e+06  2.579327e+07          46.963285     52.256469
          min    1.111209e+06  1.111345e+07           0.000000     61.000000
          25%    3.281061e+06  3.281286e+07          39.000000    116.000000
          50%    5.489399e+06  5.522408e+07          79.000000    161.000000
          75%    7.721605e+06  7.748008e+07         119.000000    207.000000
          max    9.995284e+06  9.998382e+07         188.000000    254.000000

                 nb_following    nb_tweets          age         diff
```

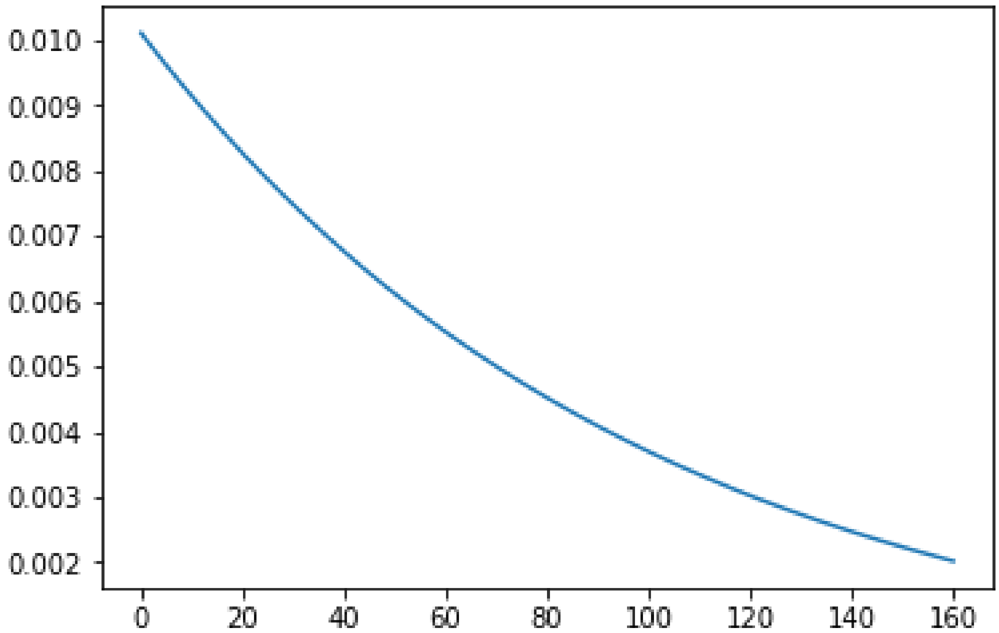
```
count   4098.000000  4098.000000  4098.000000  4098.000000
mean     141.041240    78.635920    27.497560    19.785993
std       80.418927    58.296719     6.166165    95.962284
min       32.000000     1.000000    17.000000  -447.000000
25%       77.000000    31.000000    22.000000   -39.750000
50%      126.000000    67.000000    27.000000    28.000000
75%      190.000000   114.000000    33.000000    89.000000
max      537.000000   365.000000    38.000000   218.000000
```

required_contacts mean and variance are slighty different. however, one can calculate weight a uniform constant weight IC model would have :

```
In [335]: ax = t_heavy['required_contacts'].plot.hist(bins=3, alpha=0.5)
          plt.show()
```




```
In [336]: #an probability p to infect on each edge would lead to a curve that would look like
          p = 0.01
          X = np.linspace(0,160,160)
          Y = ((1-p)**(X-1))*p
          plt.plot(X,Y)
          plt.show()
```



1.4 now let's try to segment the population in segments in which people have more homogeneous required contacts, that's to say in which people are equally likely to propagate information. We expect that demographic data yields explanatory power here because for example people in their 50s are more likely to spread fake news than millenials according to recent studies.

For the sake of interpretability, let's split the popualtion in only three groups : respectively the easy, average and hard to convince people.

```
In [414]: t_heavy[t_heavy.columns[2:]].corr()
```

```
Out[414]:                    required_contacts  nb_followers  nb_following  nb_tweets  \
          required_contacts           1.000000      0.027185      0.005135  -0.008865
          nb_followers                0.027185      1.000000     -0.001287   0.028633
          nb_following                0.005135     -0.001287      1.000000   0.001530
          nb_tweets                  -0.008865      0.028633      0.001530   1.000000
          sex                         0.006119      0.001665     -0.011988   0.014555
          age                        -0.008145     -0.000988      0.003000  -0.032153
          diff                        0.010500      0.545631     -0.838727   0.014310
          easy                       -0.707473     -0.028479     -0.010960   0.004689
          hard                        0.723824      0.006557      0.009446   0.001659
          difficulty                  0.894877      0.021939      0.012761  -0.001904

                                  sex       age      diff      easy      hard  \
          required_contacts  0.006119 -0.008145  0.010500 -0.707473  0.723824
          nb_followers       0.001665 -0.000988  0.545631 -0.028479  0.006557
```

```
nb_following      -0.011988  0.003000 -0.838727 -0.010960  0.009446
nb_tweets          0.014555 -0.032153  0.014310  0.004689  0.001659
sex                1.000000  0.002017  0.010953 -0.009200  0.017967
age                0.002017  1.000000 -0.003052  0.015315 -0.015355
diff               0.010953 -0.003052  1.000000 -0.006324 -0.004346
easy              -0.009200  0.015315 -0.006324  1.000000 -0.279020
hard               0.017967 -0.015355 -0.004346 -0.279020  1.000000
difficulty         0.016972 -0.019176  0.001253 -0.800783  0.798602

                   difficulty
required_contacts    0.894877
nb_followers         0.021939
nb_following         0.012761
nb_tweets           -0.001904
sex                  0.016972
age                 -0.019176
diff                 0.001253
easy                -0.800783
hard                 0.798602
difficulty           1.000000
```

1.5 Let's first explore wether or not demographics can help predict the sharing behavior. If it does, then we'll be able to analyse graphs on which we don't have past cascade data.

In [415]:
```python
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor

y = t_heavy.required_contacts
X = t_heavy[['nb_followers', 'nb_following', 'nb_tweets', 'age', 'sex'] ]
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)

tree_model = RandomForestRegressor(random_state = 0)
# Fit model
tree_model.fit(train_X, train_y)

# get predicted prices on validation data
val_predictions = tree_model.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))
```

```
43.6482926829
```

In [416]:
```python
from sklearn.metrics import mean_absolute_error
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
```

```
from sklearn.ensemble import RandomForestRegressor

y = t_heavy.required_contacts
X = t_heavy[['nb_tweets', 'age', 'diff', 'sex'] ]
train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)

tree_model = RandomForestRegressor(random_state = 0)
# Fit model
tree_model.fit(train_X, train_y)

# get predicted prices on validation data
val_predictions = tree_model.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))
```

```
43.6271219512
```

1.6 these models hardly do better than the target's standard deviation, so instead we'll try to classify people into the categories 'easy', 'normal', 'hard' based on the number of contact they need to spread the post.

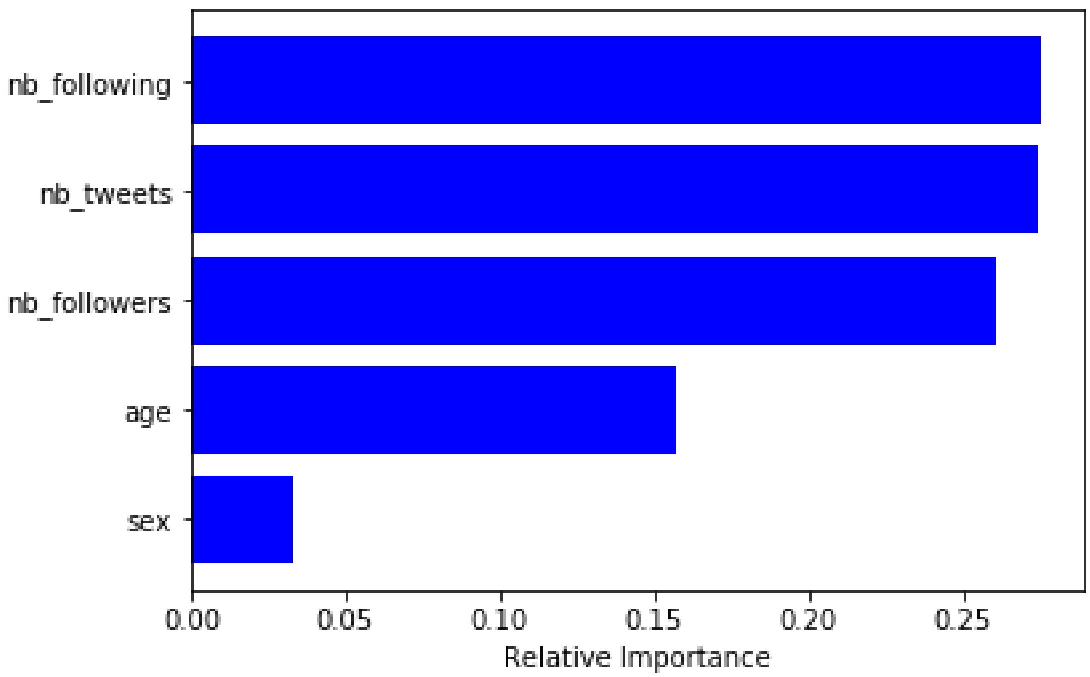
```
In [420]: t_heavy['easy'] = t_heavy['required_contacts'].apply(lambda x: x<=39)
          t_heavy['hard'] = t_heavy['required_contacts'].apply(lambda x: x>=119)
          t_heavy['difficulty'] = t_heavy['hard'].apply(int) - t_heavy['easy'].apply(int)
          t_heavy[['required_contacts','difficulty']].groupby(['difficulty']).describe()
```

```
Out[420]:           required_contacts                                            \
                                count        mean        std    min    25%    50%
          difficulty
          -1                   1033.0   20.027106  11.312991    0.0   10.0   20.0
           0                   2014.0   79.083913  22.640156   40.0   59.0   79.0
           1                   1051.0  141.189343  14.477875  119.0  130.0  140.0


                       75%    max
          difficulty
          -1          30.0   39.0
           0          99.0  118.0
           1         152.0  188.0
```

```
In [421]: from sklearn.ensemble import RandomForestClassifier
          features = ['nb_followers', 'nb_following','nb_tweets', 'age','sex']
          y = t_heavy.hard
          X = t_heavy[features]
          train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)
          clf = RandomForestClassifier(n_jobs=2, random_state=0)
          clf.fit(train_X, train_y)
          val_predictions = clf.predict(val_X)
          print(np.average(np.logical_xor(val_y, val_predictions)))
```

```
0.282926829268
```

```
In [422]: importances = clf.feature_importances_
          indices = np.argsort(importances)
          features = ['nb_followers', 'nb_following','nb_tweets', 'age','sex']
          plt.barh(range(len(indices)), importances[indices], color='b', align='center')
          plt.yticks(range(len(indices)), [features[i] for i in indices])
          plt.xlabel('Relative Importance')
          plt.show()
```

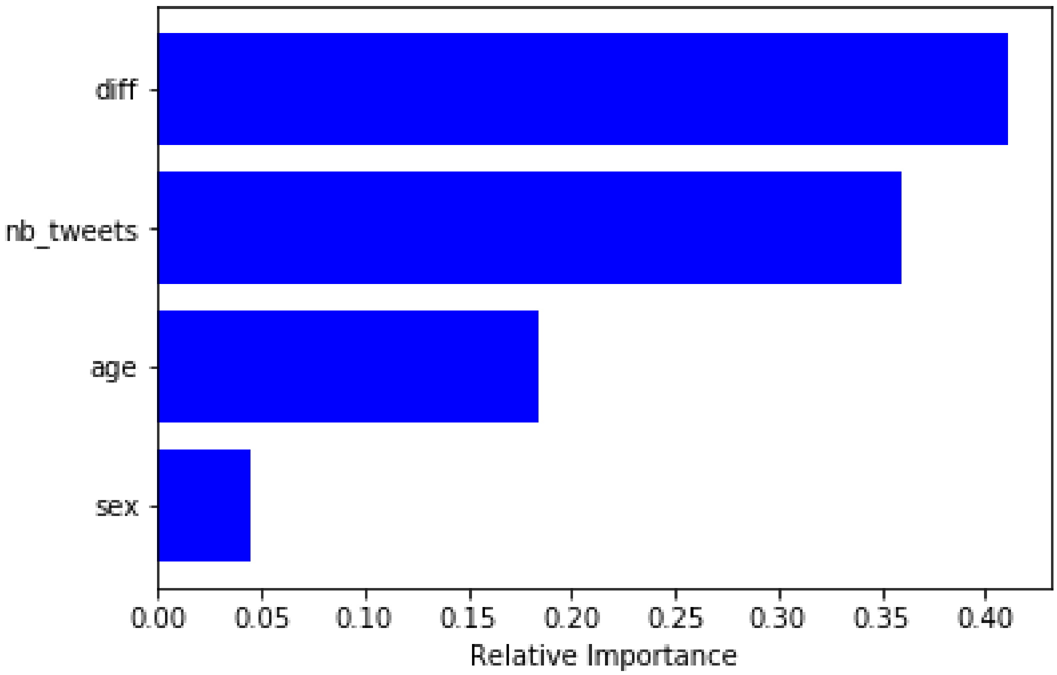



```
In [423]: from sklearn.ensemble import RandomForestClassifier
          features =  ['diff','nb_tweets', 'age','sex']
          y = t_heavy.hard
          X = t_heavy[ features]
          train_X, val_X, train_y, val_y = train_test_split(X, y, random_state = 0)
          clf = RandomForestClassifier(n_jobs=2, random_state=0)
          clf.fit(train_X, train_y)
          val_predictions = clf.predict(val_X)
          print(np.average(np.logical_xor(val_y, val_predictions)))
```

```
0.295609756098
```

```
In [424]: importances = clf.feature_importances_
          indices = np.argsort(importances)
```

```
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [features[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```




```
In [425]: np.average(val_predictions)
```

```
Out[425]: 0.087804878048780483
```

Clear underfitting : only 8% of the data is labeled as hard, while 25% of the original data is hard. Moreover, age and sex have way less importance than expected, which could be due to the non-polarizing nature of the campaign (satiric & eco-friendly).

1.7 We'll use a variation of the linear treshold model, in which the treshold of each individual (the required number of contact before it can be infected) will be chosen with gaussian distribution corresponding to that of the individual's category (hard, normal, easy) in the log data we have. Each edge has weight 1.

1.8 For users who never posted, we'll use gaussian distribution with parameters corresponding to the reunion of 'normal' and 'hard'

1.8.1 This model, which is our own, makes the assumption that even though people's behavior can't be fully understood from a single cascade, their general behavior and persona have a chance to be meaningful. This goes in the direction of classical sociological influence models in which the distinction between several segments has often been made

1.9 Let's add the instagram data first

```
In [426]: i_post, i_user = pd.read_excel("instagram_posts.xlsx", sheet_name= None), pd.read_exc
          i_post['infected_by'] = None
          for index, row in i_post.iterrows():
              id_post_origin = row['id_post_origin']
              if id_post_origin != 0:
                  infected_by = int(i_post[i_post['id_post'] == id_post_origin]['id_user'])
                  i_post.loc[index,'infected_by'] = infected_by
          columns = ['id_user','id_post', 'infected_by', 'date','time','half_day' ]
          i_lite = i_post[columns]
          for row in i_user.iterrows():
              node, neighbors = int(row[-1][0]),row[-1]['id_followers'][1:-1]
              neighbors = neighbors.split(', ')
              G.add_node(node)
              for n in neighbors:
                  n=int(n)
                  G.add_node(n)
                  elist.append((node,n))
          G.add_edges_from(elist)
          i_lite = i_lite.sort_values(by=['date', 'half_day','time'])
          vus = {}
          vect = []
          k=0
          for row in i_lite.iterrows():
              id_user = int(row[1][0])
              if k==2: print(id_user)
              k+=1
              vect.append(len([p for p in G.predecessors(id_user) if p in vus ]))
              vus[id_user]=True
          vect = np.asarray(vect)
          i_lite['required_contacts'] = vect
          i_lite.describe()
          #i_lite.head()
```

```
672702
```

Out[426]:
```
              id_user       id_post  required_contacts
count    3047.000000  3.047000e+03        3047.000000
mean   555579.649819  5.617449e+08          80.130948
std    259487.902150  2.545688e+08          47.094988
min    111274.000000  1.111918e+08           0.000000
25%    325827.000000  3.437209e+08          39.000000
50%    559608.000000  5.649181e+08          80.000000
75%    783688.500000  7.795911e+08         120.000000
max    999672.000000  9.997266e+08         292.000000
```

In [349]:
```python
t_lite['easy'] = t_lite['required_contacts'].apply(lambda x: x<=39)
t_lite['hard'] = t_lite['required_contacts'].apply(lambda x: x>=119)
t_lite['difficulty'] = t_lite['hard'].apply(int) - t_lite['easy'].apply(int)
t_lite[['required_contacts','difficulty']].groupby(['difficulty']).describe()
```

Out[349]:
```
           required_contacts                                                  \
                       count        mean        std    min    25%    50%
difficulty
-1                    1033.0   20.027106  11.312991    0.0   10.0   20.0
 0                    2014.0   79.083913  22.640156   40.0   59.0   79.0
 1                    1051.0  141.189343  14.477875  119.0  130.0  140.0


              75%    max
difficulty
-1           30.0   39.0
 0           99.0  118.0
 1          152.0  188.0
```

In [350]:
```python
i_lite['easy'] = i_lite['required_contacts'].apply(lambda x: x<=39)
i_lite['hard'] = i_lite['required_contacts'].apply(lambda x: x>=120)
i_lite['difficulty'] = i_lite['hard'].apply(int) - i_lite['easy'].apply(int)
i_lite[['required_contacts','difficulty']].groupby(['difficulty']).describe()
```

Out[350]:
```
           required_contacts                                                  \
                       count        mean        std    min    25%    50%
difficulty
-1                     765.0   19.867974  11.287165    0.0   10.0   19.0
 0                    1513.0   79.468605  23.241440   40.0   59.0   80.0
 1                     769.0  141.383615  15.211865  120.0  129.0  140.0


              75%    max
difficulty
-1           30.0   39.0
 0          100.0  119.0
 1          151.0  292.0
```

In [351]:
```python
df = pd.concat([t_lite,i_lite], axis=0)
df['time']=df["time"].astype('|S')
```

```
df = df.sort_values(by=['date', 'half_day','time'])
df.head()
```

```
Out[351]:    id_user    id_post infected_by        date         time half_day  \
          1   474227  953043456        None 2017-11-09     b'02:03'       am
          5  3003097   48168379        None 2017-11-09  b'02:53:00'       am
          2   587566  650889385        None 2017-11-09     b'02:57'       am
          2  6013435   81242015        None 2017-11-09  b'05:18:00'       am
          0   672702  638779430        None 2017-11-09     b'07:53'       am

             required_contacts  easy   hard  difficulty
          1                  0  True  False          -1
          5                  0  True  False          -1
          2                  0  True  False          -1
          2                  0  True  False          -1
          0                  0  True  False          -1
```

```
In [500]: def linear_treshold(seed, G, seuil):
              level = {k:0 for k in G.nodes()}
              for k in seed : level[k]=np.floor(seuil[k])+1
              infected = set({})
              flag =True
              while flag == True:
                  flag = False
                  for u in [k for k in level if level[k]>=seuil[k]]:
                      flag = True
                      infected.add(u)
                      level.pop(u)
                      for v in G.successors(u):
                           if v not in infected:
                              level[v]+=1
              return infected
```

```
In [457]: len(linear_treshold(list(G.nodes())[:3],G,{u:1 for u in G.nodes()}))
```

```
Out[457]: 4098
```

```
In [376]: t_lite[['required_contacts','easy']].groupby(['easy']).describe()
```

```
Out[376]:       required_contacts                                                  \
                              count        mean        std   min   25%    50%    75%
          easy
          False              3065.0  100.380098  35.748433  40.0  70.0  100.0  130.0
          True               1033.0   20.027106  11.312991   0.0  10.0   20.0   30.0


                   max
          easy
          False  188.0
          True    39.0
```

```
In [377]: t_lite[['required_contacts','difficulty']].groupby(['difficulty']).describe()
```

```
Out[377]:            required_contacts                                              \
                                 count        mean        std    min    25%    50%
          difficulty
          -1                    1033.0   20.027106  11.312991    0.0   10.0   20.0
           0                    2014.0   79.083913  22.640156   40.0   59.0   79.0
           1                    1051.0  141.189343  14.477875  119.0  130.0  140.0


                        75%    max
          difficulty
          -1           30.0   39.0
           0           99.0  118.0
           1          152.0  188.0
```

```
In [379]: i_lite[['required_contacts','difficulty']].groupby(['difficulty']).describe()
```

```
Out[379]:            required_contacts                                              \
                                 count        mean        std    min    25%    50%
          difficulty
          -1                     765.0   19.867974  11.287165    0.0   10.0   19.0
           0                    1513.0   79.468605  23.241440   40.0   59.0   80.0
           1                     769.0  141.383615  15.211865  120.0  129.0  140.0


                        75%    max
          difficulty
          -1           30.0   39.0
           0          100.0  119.0
           1          151.0  292.0
```

1.10 let's actually compute a random distribution of the tresholds following our principle of segmentation.

```
In [501]: def segmented_gaussian(G, df):
              res = {k:np.random.normal(loc=100, scale=35.74) for k in G.nodes()}
              for row in df[['id_user','difficulty']].iterrows():
                  node,diff = row[-1][:]
                  if len(str(node))==7: #dans twitter
                      if diff == -1:
                          res[node] = max(np.random.normal(loc=13, scale=13),1)
                          # we shift loc to the left to better match actual data (done after t
                      elif diff ==0:
                          res[node] = np.random.normal(loc=75.5, scale=22.6)
                      else:
                          res[node] = np.random.normal(loc=137.19, scale=14.47)
                  else:
                      if diff == -1:
```

```
                res[node] = max(np.random.normal(loc=13, scale=13),1)
                # we shift loc to the left to better match actual data (done after t
            elif diff ==0:
                res[node] = np.random.normal(loc=75.5, scale=23.24)
            else:
                res[node] = np.random.normal(loc=137.8, scale=15.21)
    return res
```

```
In [502]: def gaussian_treshold(seed, G, df):
              return linear_treshold(seed, G, segmented_gaussian(G,df))
          repeat = 5
```

```
In [503]: def gaussian_score(seed,G,df,repeat = repeat):
              return sum([len(gaussian_treshold(seed,G,df)) for k in range(repeat)])/repeat
```

```
In [504]: %%time
          print(gaussian_score(list(G.nodes())[:6], G, df, repeat=2))
```

```
1906.0
CPU times: user 2.37 s, sys: 27.5 ms, total: 2.4 s
Wall time: 2.48 s
```

At 1s per gaussian_treshold, we can't expect to compute more than tiny seedsets with the naive greedy method : we need more than 116 minutes to compute the marginal gain on each node of the 7000 nodes graph G

```
In [561]: def greedy_gaussian(G, df,k, repeat = 3):
              S = set({})
              while len(S) <k:
                  maxgain,maxnode = -1, None
                  for u in G.nodes():
                      Su = S.copy().add(u)
                      marginal_gains = []
                      for k in range(repeat):
                          reach_S = gaussian_treshold(S,G,df)
                          reach_Su = gaussian_treshold(Su, G, df)
                          marginal_gains.append(len(reach_Su.difference(reach_S)))
                      marginal_gain = sum(marginal_gains)/repeat
                      if marginal_gain > maxgain: maxgain,maxnode = marginal_gain, u
                  S.add(maxnode)
              return S
```

```
In [ ]: #gaussian_soloseed = greedy_gaussian(G, df, 2, repeat =1 )
        #takes too long
```

```
In [ ]: #soloscore = gaussian_score(gaussian_soloseed, G, df)
```

2 DISTRIBUTED CREDIT MODEL

2.1 A Data-Based Approach to Social Influence Maximization, A. Goyal

2.2 I) Scan algorithm

```
In [622]: def gamma(graph,v,u):
              return 1/graph.in_degree(u)
          def scan(graph,L,lamb):
              '''Prend en argument:
              - graph, le graphe du réseau social
              - L, le log
              - lamb, un réel qui joue le rôle de seuil de troncature
              Renvoie un dictionnaire UC tel que UC[u][v] = Gamma_{v,u}^{V-S}'''
              UC = {}
              current_table = []
              node_list = L['id_user']
              ch_order(node_list)
              parents = {}
              for v in node_list: #Initialisation de UC
                  UC[v] = {}

              for u in node_list:
                  parents[u] = []
                  for v in node_list:
                      UC[v][u] = 0

                      for v in graph.neighbors(u):
                          if v in current_table:
                              parents[u] += [v]

                      for v in parents[u]:
                          gamma = gamma(graph,v,u)
                          if gamma >= lamb:
                              UC[v][u] += gamma
                              for w in node_list:
                                  if UC[w][v]*gamma >= lamb:
                                      UC[w][u] += gamma*UC[w][v]
                  current_table += [u]
              return UC
```

2.3 II) Greedy Algorithm with CELF

```
In [623]: def computeMG(x,UC,SC):
              mg = 1
              for u in node_list:
                  if UC[x][u] > 0:
                      mg += UC[x][u]
              return mg*(1 - SC[x])
```

```
def update(x,UC,SC):
    for u in node_list:
        if UC[x][u] > 0:
            for v in node_list:
                if UC[v][x] > 0:
                    UC[v][u] -= UC[v][x] * UC[x][u]
            SC[u] += UC[x][u]*(1-SC[x])
def greedy(UC,k,L):
    SC = []
    S = []
    Q = []
    node_list = L['id_user']
    for u in node_list:
        mg = computeMG(u,UC,SC)
        it = 0
        Q = [(u,mg,it)] + Q
    while len(S) < k:
        (x,mg,it) = Q.pop()
        if it == len(S):
            S.append((x,mg,it))
            update((x,mg,it),UC,SC)
        else:
            mg = computeMG(x,UC,SC)
            it = len(S)
            Q = [(x,mg,it)] + Q
    return S
```

a previous run at k = 2 gave the following result :

```
In [638]: credit_seed = {2:{201595,131644}}
```

```
In [625]: gaussian_score(seed=credit_seed[2],df=df,G=G,repeat=10)
```

```
Out[625]: 2501.5
```

Because of the long computation time, we can't test the credit seed on k > 2. Still, it performs very well, exceeding the score of k_core.

3 COMPARISON

3.1 now let's get seed sets of various sizes via PageRank and k-Core decomposition of G for purposes

```
In [602]: %%time
          k_core_seed = {}
          pagerank_seed = {}
          degree_seed = {}
          #G.remove_edges_from(nx.selfloop_edges(G))
```

```
def keywithmaxval(d,n):
    v=list(d.values())
    k=list(d.keys())
    max_keys = set()
    for i in range(n):
        max_keys.add(k[v.index(max(v))])
        v[v.index(max(v))]=0
    return max_keys
for k in range(30):
    degree_seed[k] = keywithmaxval(nx.degree_centrality(G),k)
    pagerank_seed[k] = keywithmaxval(nx.pagerank(G),k)
    k_core_seed[k] = keywithmaxval(nx.core_number(G),k)
```

```
CPU times: user 5min 40s, sys: 7.42 s, total: 5min 47s
Wall time: 5min 52s
```

In [603]:
```
print('out of 10 in the k_core seed set, ',len([k for k in k_core_seed[10] if len(st
# only instagram users : instagram is denser :
#people tend to adopt a follow/follow back behavior which is not the case on Twitter
```

```
out of 10 in the k_core seed set,  0  users belong to twitter
```

In [604]:
```
print('out of 10 in the degree seed set, ',len([k for k in degree_seed[10] if len(st
```

```
out of 10 in the degree seed set,  5  users belong to twitter
```

In [512]:
```
print('out of 10 in the pagerank seed set, ',len([k for k in pagerank_seed[10] if le
```

```
out of 10 in the pagerank seed set,  6  users belong to twitter
```

In [605]:
```
gaussian_score(k_core_seed[10], G, df, repeat=5)
```

Out[605]: 3284.8

In [477]:
```
gaussian_score(k_core_seed[2], G, df, repeat=5)
```

Out[477]: 1651.6

In [606]:
```
gaussian_score(degree_seed[10], G, df, repeat=5)
```

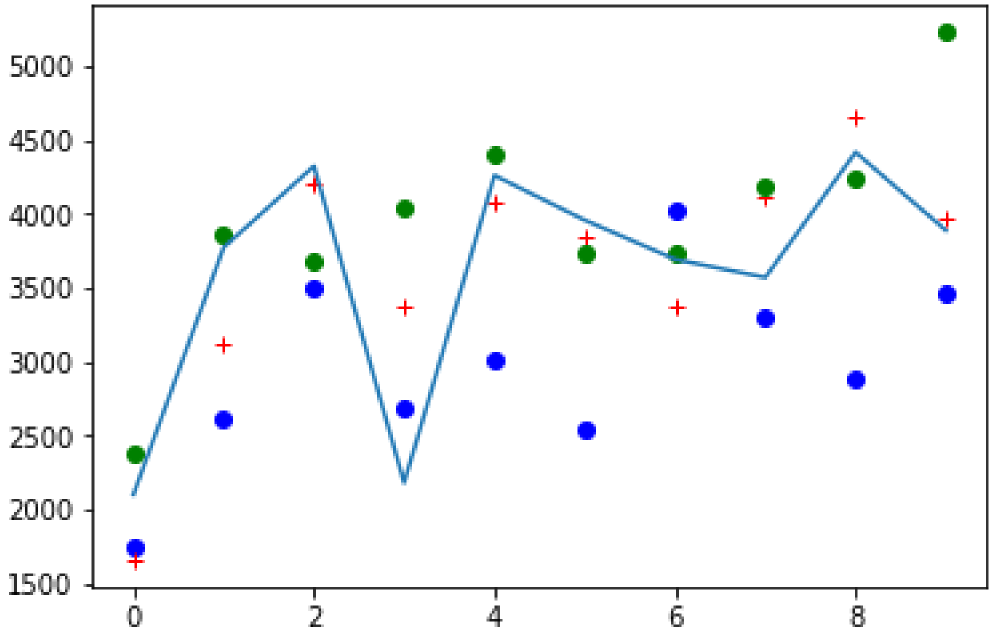
Out[606]: 4012.6

In [479]:
```
gaussian_score(degree_seed[2], G, df, repeat=5)
```

Out[479]: 4011.2

In [628]:
```
gaussian_score(pagerank_seed[10], G, df, repeat=5)
```

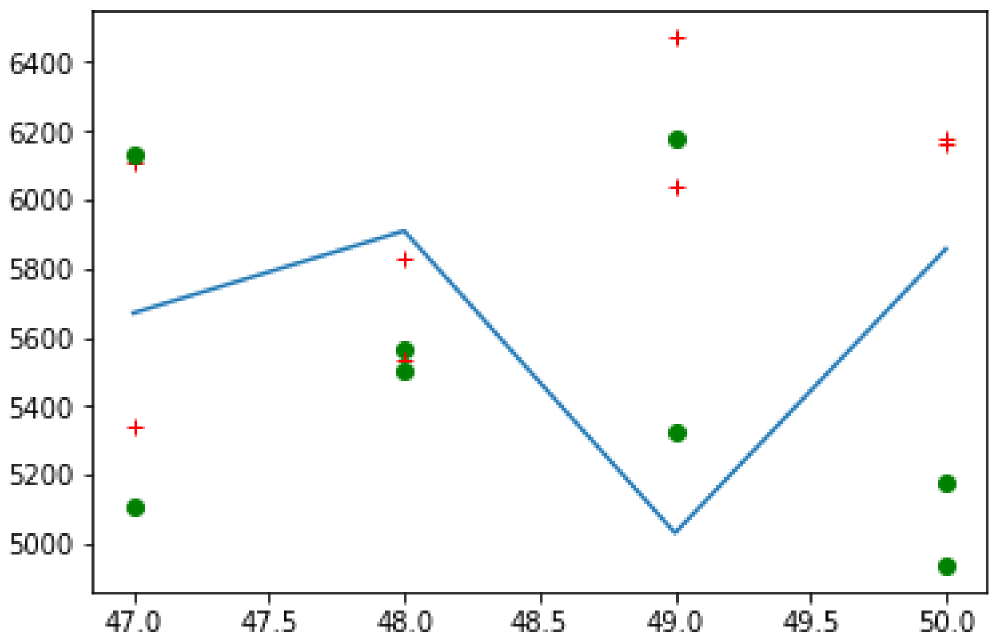
```
Out[628]: 3087.6

In [627]: gaussian_score(pagerank_seed[2], G, df, repeat=10)

Out[627]: 3656.1

In [607]: %%time
          X = np.array([k for k in range(10)])
          Ycore = np.array([gaussian_score(k_core_seed[k],G,df, repeat = 10) for k in range(10
          Ydegree = np.array([gaussian_score(degree_seed[k],G,df, repeat = 10) for k in range(1
          Ypagerank = np.array([gaussian_score(pagerank_seed[k],G,df, repeat = 10) for k in ran
          Yrandom  = [gaussian_score(np.random.choice(list(G.nodes()),size=k),G,df, repeat = 10

CPU times: user 7min 9s, sys: 1.5 s, total: 7min 11s
Wall time: 7min 10s


In [608]: plt.plot(X,Ycore,'bo') #blue circles
          plt.plot(X,Ydegree, 'r+') #red crosses
          plt.plot(X,Ypagerank, 'go') # green circles
          plt.plot(X,Yrandom)
          plt.show()
```



any of these seedsets hardly do better than random sampling for k<10. pagerank looks the most promising, let's see how he competes for k =50.

```
In [633]: %%time
          for k in range(47,51):
```

```
    degree_seed[k] = keywithmaxval(nx.degree_centrality(G),k)
    pagerank_seed[k] = keywithmaxval(nx.pagerank(G),k)
X = np.array([k for k in range(47,51)])
Yrandom  = [gaussian_score(np.random.choice(list(G.nodes()),size=k),G,df, repeat = 10
Ydegree = np.array([gaussian_score(degree_seed[k],G,df, repeat = 10) for k in range(4
Ypagerank = np.array([gaussian_score(pagerank_seed[k],G,df, repeat = 10) for k in ran
plt.plot(X,Ydegree, 'r+') #red crosses
plt.plot(X,Ypagerank, 'go') # green circles
plt.plot(X,Yrandom)
plt.show()
```




```
CPU times: user 3min 11s, sys: 2.37 s, total: 3min 14s
Wall time: 3min 23s
```

It appears that this model doesn't discriminate much between seedsets, du to the high degree of randomness.

```
In [635]: %%time
          gaussian_score(pagerank_seed[50],G,df,repeat = 50)
```

```
CPU times: user 1min 1s, sys: 425 ms, total: 1min 2s
Wall time: 1min 2s
```

```
Out[635]: 5279.5
```

I'd argue that the best pick would be to use pagerank up until we exhaust the budget; but it's remarkable that it's not ignificantly better than random choice. Most probably this is due to the randomness of the model, which is too high. If we had more time, we'd reduce the std's of the segment in order to make a more distinct separation and reduce randomness. However, let's not forget that real-world propagation is highly random, and such a model isn't irrelevant. What could be a more sound idea would be to dramatically increase the repeat parameter in order ro reduce variance of results.

```
In [636]: gaussian_score(pagerank_seed[50],G,df,repeat = 100)

Out[636]: 5319.27
```

4 Alternative COMPARISON

4.1 data driven comparison : DNI (distinct node infected)

see DiffuGreedy: An Influence Maximization Algorithm based on Diffusion Cascades, G. Panagopoulos, F. Malliaros, M. Vazirgiannis, 2018 ## The DNI of a seed set is the number of distinct nodes it would have infected in the past cascade that we have.

An important feat of DNI is that it relies only on past data, not on the inputed structure of the graph, and is devoid of probabilistic data or simulations.

```
In [609]: '''
          Input: Log (t_post) and the seeds (seed_set)
          Output: the DNI list
          Measures how good are the seeds (seed_set) chosen given the trace (log) = t_post

          '''
          def get_dni(df, seed_set, G=G):
              log = df.copy()
              DNI = set({})
              if seed_set == set(): return set()
              to_visit = list(seed_set)
              while len(to_visit)>0:
                  x = to_visit.pop()
                  if x not in DNI:
                      for infected_by_x in list(log.id_user[log['infected_by']== x ]):
                          if infected_by_x not in DNI:
                              to_visit.append(infected_by_x)
                      DNI.add(x)
              return DNI

In [540]: len(get_dni(df, set({})))

Out[540]: 0

In [535]: get_dni(df, [5662813]) # le noeud seed 5662813 n'a propagé l'information qu'a une se

Out[535]: {2152321, 5662813}
```

```
In [536]: len(get_dni(df, [3003097,6013435,6027974,1953787,9834565,3027418])) #le seed entier
```

```
Out[536]: 4098
```

```
In [610]: def dni_score(df, seed_set):
              return len(get_dni(df, seed_set))
```

```
In [441]: dni_score(df, [3003097,6013435,6027974])
```

```
Out[441]: 1735
```

```
In [616]: def greedy_dni(df,k):
              S = set({})
              while len(S) <k:
                  maxgain,maxnode = -1, 3003097
                  reach_S = get_dni(df,S)
                  for u in G.nodes():
                      Su = S.copy()
                      Su.add(u)
                      reach_Su = get_dni(df,Su)
                      marginal_gain = len(reach_Su.difference(reach_S))
                      if marginal_gain > maxgain: maxgain,maxnode = marginal_gain, u
                  S.add(maxnode)
              return S
```

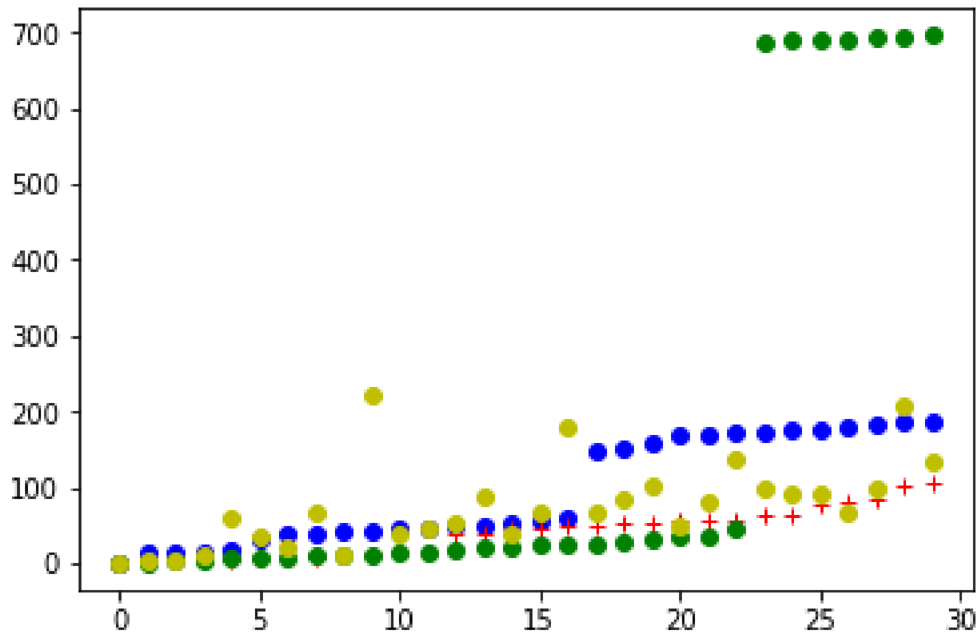
```
In [631]: %%time
          #dni_score(df,greedy_dni(df,2))
```

```
CPU times: user 5 ţs, sys: 1 ţs, total: 6 ţs
Wall time: 11 ţs
```

```
In [619]: dni_score(df,greedy_dni(df,1))
```

```
Out[619]: 1100
```

```
In [621]: %%time
          X = np.array([k for k in range(30)])
          Ycore_dni = np.array([dni_score(df,set(k_core_seed[k])) for k in range(30)])
          Ydegree_dni = np.array([dni_score(df,set(degree_seed[k])) for k in range(30)])
          Ypagerank_dni = np.array([dni_score(df,set(pagerank_seed[k])) for k in range(30)])
          Yrandom = [dni_score(df,set(np.random.choice(list(G.nodes()),size=k))) for k in rang
          plt.plot(X,Ycore_dni,'bo') #blue circles
          plt.plot(X,Ydegree_dni, 'r+') #red crosses
          plt.plot(X,Ypagerank_dni, 'go') #green circles
          plt.plot(X,Yrandom, 'yo') #yellow circles
          #plt.plot(X, Ygreedy_dni)
          plt.show()
```




```
CPU times: user 20.8 s, sys: 249 ms, total: 21 s
Wall time: 21.1 s
```

for small k, our crafted seeds performs hardly better than random sampling. However when k goes past 15 there is a clear enhancement of such seedsets. The greedy algorithm is somewhat too slow to be compared here but it is way better (1100 for k=1 (optimal) !) There again, among traditionnal methods pagerank clearly dominates the others and would make the best data-blind pick, with a critical initial mass of 16. Of course greedy is better, but greedy_dni was designed to beat the log cascade and is thus overfitted to the cascade by design. What we derive from that is that we need more data in order to make DNI significant. Indeed, DNI based methods will necessarily pick the initiators of the cascade we have and then stagnate due to the 0 marginal gain of subsequent nodes. As a rule of thumb, we need way more cascades than the size of the seedset (10 times at least), in order for the algorithm to be useful (not simply return one of the few initiators, and not stagnating after k > number of cascades). Because of how credit flows in the credit distribution model, the same remarks applies to this model.

4.1.1 And what about credit ?

We expect credit to work fine, as with only 1 cascade, maximizing credit is analog to maximizing DNI

4.2 Marketing value

To be more helpful in a real-word application, we propose here a modified version of the greedy gaussian treshold algorithm, which takes into account actual marketing strategies instead of a constant seed set size

In this model, there is a different price p1,p2,p3 which corresponds to easy, normal and hard to convince segments of the population. Indeed, one has to display more ads, or ads of better quality, to convince 'hard' people that they should share the information.

This algorithm maximizes $\frac{marginalgain}{price}$ instead of $marginalgain$ alone and stops when the remaining budget is less than the price of any segment.

```
In [ ]: def segment(u):
            return 1 # supposedly returns the segment of u
        def greedy_gaussian_marketing(G, df,k, prices = {'easy':0.75, 'normal':1,'hard':1.25},
            S = set({})
            while len(S) <k:
                maxgain,maxnode = -1, None
                for u in G.nodes():
                    Su = S.copy().add(u)
                    marginal_gains = []
                    for k in range(repeat):
                        reach_S = gaussian_treshold(S,G,df)
                        reach_Su = gaussian_treshold(Su, G, df)
                        marginal_gains.append(len(reach_Su.difference(reach_S)))
                    marginal_gain = sum(marginal_gains)/(repeat*price[segment(u)])
                    if marginal_gain > maxgain: maxgain,maxnode = marginal_gain, u
                S.add(maxnode)
            return S
```

5 Conclusion and results

5.1 because of how well it performed in the two models, I'd go for a pagerank seedset of suitable size (budget-wise). Both models show that there is only little improvement after size 16, which is perfectly sound because of the submodularity of the underlying function in both models.

5.1.1 Greedy algorithms would provide with the bst picks but that would require more computational ressources and/or more data

6 To go further

The bottleneck here is either the long computation time in the case of the gaussian greedy algorithm that we introduced or the lack of cascade data, with more of which we could make better use of the DNI algorithm.

DeepInf: Social Influence Prediction with Deep Learning Jiezhong Qiu , Jian Tang , Hao Ma , Yuxiao Dong , Kuansan Wang , and Jie Tang

Deep neural networks : we are capable of generating infinitely many social networks and run smulations on them -> infinite training data but the bottleneck is computation time

Marketing-wise, it is desirable to go further and stop assuming that anyone can be bought in the beginning, and instead of looking for a seed_set, we would be looking for an ordonned set of all nodes with respect to their priority, and our client could just try and 'buy' them with respect to that order up until the budget is exhausted

An alternatve to this is to determine marketing strategies, for exemple by segmenting the population, and then maximizing the function g(x) as defined in Maximizing the Spread of Influence through a Social Network, D.Kempe; by hill-climbing methods.

6.1 External References

A Data-Based Approach to Social Influence Maximization, A. Goyal

DiffuGreedy: An Influence Maximization Algorithm based on Diffusion Cascades, G. Panagopoulos, F. Malliaros, M. Vazirgiannis, 2018

Maximizing the Spread of Influence through a Social Network, D.Kempe

DeepInf: Social Influence Prediction with Deep Learning, J. Qiu, J. Tang, H. Ma, Y. Dong, K. Wang and J. Tang