RECOMMENDER SYSTEM FOR BEERADVOCATE

SMM638 NETWORK ANALYTICS FINAL COURSE PROJECT

Group 3
Ching-Wei Liu
Fangyuan Zhao
Minghua Tu
Ruiqi Huang
Yu Bian

13 December 2018

Contents

1 Introduction

1.1 Project Objectives

BeerAdvocate.com is one of the largest and most diverse online communities for beer enthusiasts to review, support and promote better beer. In this project, we aim to build a recommender system for BeerAdvocate website in order to improve its financial situation by increasing website traffic. In particular, we plan to achieve this by building a beer recommender system which will be able to

- motivate occasional users to engage with the platform more frequently and actively;
- encourage heavy users to discover portions of the platforms which they are less familiar with.

We try to approach this problem from the perspective of network science by three different recommending algorithms:

(i) by community,

(ii) by hubs,

(iii) by neighbours' neighbour.

The theory behind each method will be elaborately explained in section 2, and the technical details will be discussed in section 3.

1.2 Recommender System Overview

According to Resnick & Varian (1997), "Recommender Systems are software tools and techniques providing suggestions for items to be of use to a user". People nowadays intentionally or unintentionally leave behind a massive amount of digital footprints every seconds of every day, resulting in large volumes of data which can be analysed and utilised by interested parties (with authorisation) (Fish 2009). Therefore, the topic of recommender systems has become increasingly trendy in recent years. Recommender systems are used in a wide variety of areas, such as music, movies, retail, news, online dating, friends on social media, etc. In this project, we will use it to recommend beers for beer enthusiasts on the website BeerAdvocate.com.

Generally, there are two types of approaches to build a recommender system: Content-Based Filtering and Collaborative Filtering.

- **Content-Based Filtering** focuses on the similarities between different products. When it calculates their similarity, it takes many attributes of the products into consideration, then it recommends the product which has the most common features (Lops et al. 2011). In our case, it would consider the ingredients, taste, smell, look, origin of brewery, alcohol level, notes, price, etc. of a given beer product.
- **Collaborative Filtering**, on the other hand, does not require any actual knowledge about the products. Instead, it exploits users' information to build the recommender model. Specifically, it can be further divided into two categories:

  (i) *Item-Based Collaborative Filtering* (IBCF) analyses historical data in order to find relation between co-purchased products (i.e. whether the purchase of one product often leads to the purchase of another product) (Sarwar et al. 2001). Once this relation is found, the system then produces the recommendation based on the strength of association between products.

(ii) *User-Based Collaborative Filtering* (UBCF) aims to identify a community of users who have previously exhibited similar behaviour or had a history of agreeing with each other (Karypis 2001, Sun et al. 2015), e.g. commented on the same post, liked/disliked the same (type of) product. Once the community is detected, the system produces recommendation based on the combined preferences of members in the community.

In this project, we will put more focus on UBCF in order to make the most out of the theory we have learnt in SMM638 Network Analytics module, as well as some other recommending algorithms .

2 Theoretical Methodology

2.1 Recommending by Community (UBCF)

The first recommending algorithm aims to motivate users engaging more with the platform, we implement the *User-Based Collaborative Filtering* algorithm to predict users' interests by detecting what community they belong to, then recommend them the communities' favourite beers.

It is supported by the concept of *homophily*. McPherson et al. (2001) states that " homophily is the principle that a contact between similar people occurs at a higher rate than among dissimilar people." In our case, it can be argued that since a user is belong to a particular community, it is fairly likely that he/she would like the beer that also liked by other community members.

Network Formation

In order to detect communities from our network, we first need to identify the network from our raw data `all_user_interactions`. In particular, we consider information about users and their activities in forums.

A user responding to a post creates a tie between them. Then a two-mode unweighted network is formed, as shown in the left graph of Figure 1. We have a set of green nodes representing users and a set of red nodes representing forums. Since we are modelling relations between two different types of objects, a *bipartite graph* naturally arises. According to Barabási et al. (2016), "a bipartite graph is a network if its nodes can be divided into two disjoint sets U and V such that each link connects a U-node to a V-node", meaning that a tie can only exist between two objects of different types.

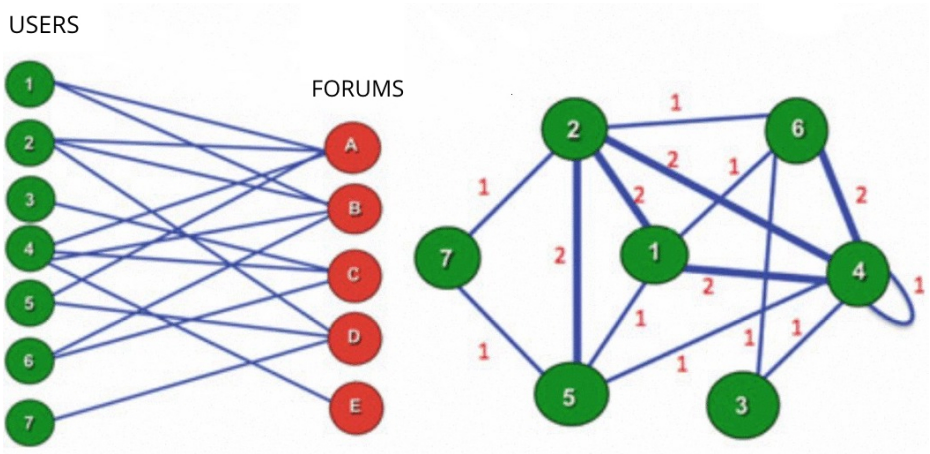

Figure 1: An example of weighted bipartite projection. (Alzahrani & Horadam 2016)

We consider that a tie exists between two users if they have interacted with the same forum. For example, in the left graph of Figure 1, there are ties between forum C and each of users 3, 4, 6, meaning all of the three users are active in forum C. Users 3, 4, 6 are therefore connected to each other. Additionally, users 4, 6 are also both tied with forum B, resulting in a tie between users 4 and 6 with weight = 2 (they both responded with two different forums). Same thinking can be applied to all other nodes. Repeating this procedure, we have projected the two-mode unweighted network between users and forums to a one-mode weighted network of users, as shown in the right graph of Figure 1.

Community Detection

Now we are ready to detect communities in this one-mode weighted network of users in Python, and recommend the community favourite beers to its members. The traditional way to detect communities is to use the *Girvan-Newman algorithm*.

Girvan-Newman algorithm is a hierarchical method which detects communities by progressively removing edges with the largest *betweenness* (number of shortest paths passing through an edge) (Girvan & Newman 2002). However, since we are dealing with several rather large datasets, it is extremely slow to actually use the Girvan-Newman way.

Therefore, we will instead use the function `cluster_walktrap` from `igraph` to perform community detection. This function is based on the *Latapy & Pons algorithm*, which is built on the idea of *random walks* and is "efficiently computable for large and complex networks" (Pons & Latapy 2005). Interested readers are encouraged to check out their original paper for more detail.

2.2 Recommending by Hubs

The second recommending algorithm aims to encourage users to discover new beers, we decide to incorporate some *randomness* into our UBCF system by identifying *hubs*. Essentially, hubs can be thought as influencers, who interact extensively with the network in comparison to normal users (Rob Parkin 2004). Such interactions include posting articles, sending & receiving likes, commenting, etc., and each interaction adds a tie between the influencer and another user. Once an influencer has been identified, we then recommend his/her preferred beers to all users in the network.

In network science, "a hub is a node with a number of links that greatly exceeds the average" (Barabási et al. 2016). Hubs emerge in *scale-free* networks, which are networks whose degree distribution follows a power law (Barabási et al. 2016). One of the features of the power law distribution is that only very few nodes have very high degree, and these nodes would be our hubs. We identify hubs by calculating the *betweenness centrality* (the number of these shortest paths that pass through the node) of each node in the network.

There are two advantages of this algorithm. Firstly, it provides heavy users with a fresh set of beer suggestions that are different from their current preferences, thereby encouraging them to jump out of their "beer comfort zones". Secondly, the UBCF algorithm alone might not work effectively for occasional users who have yet to belong to any community. Recommending by hubs, however, is able to provide recommendation to such occasional users despite the lack of social relationship.

2.3 Recommending by Neighbours' Neighbour

The third algorithm is designed specifically to provide beer recommendation for occasional users on BeerAdvocate. As mentioned in the last section, since they might not be experienced enough

to belong to any particular community, we could use hubs to generate recommendations for them. Another appropriate algorithm is based on the preference of the neighbours' neighbour of an occasional user. To be more specific, if a user A is connected to user B, and user B is further connected to user C, we could then recommend C's preferred beers to A. This algorithm preserves similarity while delivers randomness to a certain extant.

The theory behind this algorithm is the concept of *triadic closure*. According to Watts (2004), "Triadic closure is the property among three nodes A, B, and C, such that if a strong tie exists between A-B and A-C, there is a weak tie between B-C." This proves that a weak tie exists between this user and his/her neighbour's neighbour, and it is reasonable to recommend beers based on this weak tie.

2.4 Recommending Mechanism

We categories users into occasional and heavy user groups. *For the simplicity and clarity of this project, we assume that heavy users are experienced enough to belong to communities, and occasional users are not yet belong to any communities, or they have not rooted deeply enough in their communities so that their preferences would be affected by the communities preferences.*

Based on this assumption, we will recommend beers to heavy users using the first and the second algorithms with weights 50% and 50%. This means that in our list of recommendations, half of them comes from the first algorithm, and the other half comes from the second. This ensures equal levels of similarity and randomness, allowing heavy users to jump out of their local communities. For occasional users, we shall implement the second and the third algorithms with equal weights based on the similar reasoning.

3 Practical Implementation

In this section we will explain in detail how we clean the data, implement each algorithm, and create Python functions accordingly. Intuitively, we aim to create a function which requires an user ID as an input, and its output will be a list of 10 beers to recommend to this particular user.

3.1 Differentiating Occasional and Heavy Users

Idea

When we have a user ID on hand, in order to decide which recommending algorithms to use, the first thing to do is to determine whether it is a heavy or an occasional user.

First we need to clarify what are "heavy" and "occasional" users. For each user, we define a new parameter "activeness" to be the average number of interactions per hour. More specifically, activeness equals to the number of interactions (posts, comments, etc.) of the user on BeerAdvocate forums divided by the user's tenure (how long he/she has joined BeerAdvocate, in hours).

$$\text{activeness} = \frac{\text{number of interactions}}{\text{tenure}}$$

[Note: we have double checked the fact that all tenure values are greater than 1, otherwise this activeness formula would be invalid.]

A larger activeness coefficient indicates a higher level of engagement of a user. So we sort this value for all users in descending order, then we define heavy users to be users with activeness levels at the upper third out of all users, and occasional users to be the rest (lower two thirds).

Working Process

Step 1: In order to find the total activities, we extract user ID from original dataset `all_user_interactions.csv`. We proxy the total activity of each user by their total number of posts or threads, which is indicated in the `var` column of the dataset. By grouping the user ID, we calculate the total activities of each user and assign the user ID and their corresponding activities into a new dataframe.

Step 2: The tenure (joining time) of each user can be found in dataset `gb_beer_reviewers_attributes.csv`. Since the column indicating joining time also contains information of `last_activity`, we need to subset the dataset containing only joining time information. After slicing the data, the next step is to translate the joining time format, from `string` to `pd.Timestamps`. We found some special value of the joining time such as "Yesterday", "Wednesday", etc, therefore dropping the special values is a must before converting the data. After the cleaning and translating has been done, the last step is to calculate the tenure based on the users joining time. Tenure, being the difference between joining time and the present, is stored in a new column as type using function `astype('timedelta64[h]')`.

Step 3: We now have the two ingredients of the formula and is ready to calculate the desired activity level. Merge the two datasets we got from previous steps and now we have the final dataframe containing `user ID`, `total activity` and `tenure`.

Step 4: We have finalised all the preparation work and is ready to create the profile function. The function needs two inputs, namely user ID and the final dataframe. We apply the formula to calculate the activity level and store it into a new column into the final dataset. We use `np.percentile()` to calculate the threshold of heavy/occasional users, following by a for loop categorizing users row by row and writing the value into a new column called `category` of the final dataframe. Based on the dataframe, we extract specific value from `category` based on the user ID provided and return as output. Bingo!

Key code for generating this function is shown in Fig.2

```
def profile(usr_id,usr_level):
    # set the formula of activity level which is total activity divided by tenure
    usr_level.loc[:,'level'] = usr_level.activity/usr_level.tenure
    # define users whose activity level is within the top 1/3 as 'heavy users', otherwise 'occational users'
    usr_level.loc[:,'category'] = ''
    threshold  = np.percentile(usr_level['level'], 66.67)
    a = usr_level['level']

    l = []

    for i in range(len(a)):
        if a[i] >= threshold:
            l.append(1)
        else:
            l.append(0)

    usr_level['category'] = l
    category = usr_level[usr_level.usr == usr_id].category.item()

    return category
```

Figure 2: Dividing users into two groups

3.2 By Community

Idea

As mentioned in section 2.1, we detect communities in our users interaction network using the Latapy & Pons algorithm. When the previous step determines a heavy user, it will then go through this recommending function.

It first tests which community this user belongs to, then it identifies a list of top 25% rated beers in this community. Specifically, we collect rating of each individual beer product from each community member and calculate the average rating for this beer in the community, then we sort the average ratings in descending order, and catch the first 25% of them.

After we have a list of candidate beers, we randomly select 5 beers from the list. This is then our recommendation for this user based on his/her community's.

The reason why we do not simply recommend the top 5 beers to every community member is that there would a fairly big chance that many members already have tried these beers because of how popular they are. Such a recommender system would be much less interesting and less helpful for heavy users.

Working Process

Step 1: Convert the edge list in dataframe `usr_usr` into graph g using igraph. Referring to igraph documentation, we use `cluster_walktrap` function to cluster the communities within the original network, and use a for loop to store communities into a dictionary with the key to be the name of the communities and the value to be the corresponding user ID.

Step 2: Define a function in which if we input a user ID, the output is the community ID which the user belongs to. We apply the function to each user ID within the network and create a new column in dataframe `beer_rev` containing the community information for each user. As the unique users in dataframe `usr_usr` is different from that in dataframe `beer_rev`, the reason being that there might be users who posted on forums but have not reviewed any beer or otherwise. Therefore, while we apply the function to users that do not exist in `beer_rev`, the community ID returned could be `NA`. We then drop these `NA` values.

Step 3: We find the beers reviewed by each community. The average score of a beer is used as an indicator of its popularity. Thus, we calculated the average score of beers reviewed by collapsing the dataset by both user ID and communities, and calculated the average score of beers reviewed for each community.

Step 4: We extract the top beers from each community. The selection criteria are to select beers in the first quartile (top 25%) of score distribution in a community. In order to do this we write a for loop to populate the selected beers into a dictionary which keys are community IDs.

Step 5: Using the function we defined before to find the unique user IDs community and using the dictionary we defined just now to find their top beers, we return the result as output and finish the whole function.

Key code for generating this function is shown in Fig.3

```
# extract top beers from each community
community_top = {}
for k in community.items():
    subset = beer_com.loc[beer_com['com'] == k[0]]
    list = []
    for index, row in subset.iterrows():
        # select beers in the first quantile of score distribution in a community
        if row['bascore_norm']>= max(subset['bascore_norm']*0.75):
            list.append(row['beer'])
    community_top.update({k[0]:list})

# find the community user belongs to
membership = find_key(community,usr_id)
return community_top.get(membership)
```

Figure 3: Finding top beers in each community

3.3 By Hubs

Step 1 We use the file `user_user_weighted_graph.csv`, which was generated by the bipartite graph projection step, to create an edge list showing the source user, the target user and the weight of their tie.

Step 2 Since the BeerAdvocate user network is a large network, we choose to use another python module `igraph` instead of `NetworkX` to generate the graph of the weighted network of BeerAdvocate users. As a result, we apply the function of `TupleList` in `igraph` module over the edge list generated in Step 1 to create the users network graph.

Step 3 Hubs are defined as users who have higher shortest-path betweenness centrality. In this step, we apply the `betweenness` function in `igraph` to find the shortest-path betweenness centrality of each user and then selected the top 100 users as our hubs.

Step 4 After we got the list of hubs, we merge this data with the dataframe created by the file `gb_beer_reviews.csv` to get a dataframe stating all the beers reviewed by hubs.

Step 5 Since the beer preference of users changes as time goes on, we only choose the beers which are most recently reviewed (in the last 3 months) and are rated higher (score is greater than 4) from the dataframe created in Step 3.

Step 6 Finally, we randomly selected 20 beers from the dataframe of beers reviewed by hubs, which is updated in Step 5, as the list of beers to be recommended to users.

Key code for generating this function is shown in Fig.4

3.4 By Neighbours' Neighbour

Step 1 Obtain the user ID of source nodes and target nodes from `user_user_graph.csv` by splitting the string that contains both source and target nodes.

Step 2 Generate a network with the data frame containing both source and target nodes.

Step 3 We find a list of the neighbour of the target user that is selected. At this step, we have a large number of neighbours, and then we apply a for loop to loop through all the neighbours to find the neighbour's neighbour. This will generate a list of source node's neighbour's neighbour.

```
#%% Find hubs

# Calculate top 100 betweenness centrality of user network
btvs = []
for p in zip(user_user_weighted_network.vs, user_user_weighted_network.betweenness()):
    btvs.append({"name": p[0]["name"], "bt": p[1]})
user_user_weighted_bc = sorted(btvs, key=lambda k: k['bt'], reverse=True)[:100]

# Get the hub list
hub_nodes_list = []
for i in range(len(user_user_weighted_bc)):
    hub_nodes_list.append(user_user_weighted_bc[i].get('name'))

#%% The beers reviewed and scored by the hubs

beer_rtdate = beer_usr_rtdate

# slice beers reviewed in last 3 months
beer_rtdate['date'] = pd.to_datetime(beer_rtdate['date'], errors='coerce')
beer_rtdate['year'] = beer_rtdate['date'].dt.year
beer_rtdate['month'] = beer_rtdate['date'].dt.month
beer_rtdate = beer_rtdate[beer_rtdate.year == 2018]
beer_rtdate = beer_rtdate[beer_rtdate.month >= 9]

# slice beers scored >= 4
beer_rtdate = beer_rtdate[beer_rtdate.bascore_norm >= 4]

# Prepare beer data for hubs
hub_nodes_df = pd.DataFrame(hub_nodes_list, columns=['hub'])
hub_nodes_df.rename(columns={"hub": "usr"}, inplace=True)

hub_beers_df = hub_nodes_df.merge(beer_rtdate,on='usr')
hub_beers_df = hub_beers_df.drop_duplicates('beer')

# Randomly selected from hub beers
hub_beers_sample = hub_beers_df.sample(n = 20)

# Generate the hub beers list
hub_beers_list = hub_beers_sample['beer'].tolist()
```

Figure 4: Recommending by hubs

Step 4 We transform the list of neighbour's neighbour from Step 3 into a dataframe and merge it with reference of the user ID to the `gb_beer_reviews.csv` to find the list of beers that the neighbour's neighbour has rated before.

Step 5 After we have the data frame of beers for neighbor's neighbor, we have introduced a concept of popularity (see 1) with max score of 400 into the dataframe and merge it with reference of beer ID, so we have assigned each beer with a popularity score.

$$\begin{aligned}
\text{if } X > 100 &: 200 + \frac{\text{taste score}}{5} \times 100 + \frac{\text{smell score}}{5} \times 100; \\
\text{if } 20 < X \leq 100 &: 150 + \frac{\text{taste score}}{5} \times 100 + \frac{\text{smell score}}{5} \times 100; \\
\text{if } 5 < X \leq 20 &: 100 + \frac{\text{taste score}}{5} \times 100 + \frac{\text{smell score}}{5} \times 100; \\
\text{if } 0 < X \leq 5 &: 50 + \frac{\text{taste score}}{5} \times 100 + \frac{\text{smell score}}{5} \times 100.
\end{aligned} \tag{1}$$

Step 6 We sort the dataframe according to the popular score in descending order, and extract the top 3 beers with the highest popularity score from the data frame and then select another three beers randomly from the data frame excluding the top 3 beers, by this, we could avoid duplication of beers.

Key code for generating this function is shown in Fig.5.

```
#Find top_3 beer according popularity
usr_NN_beer = usr_NN_beer.sort_values(by=['popular'], ascending= False)

#select the top 3 beers for neighbor's neighbor
top3 = usr_NN_beer.iloc[0:3, :]

#crop the data without the top 3 beers to avoid duplication
df_no_top3 = usr_NN_beer.iloc[3:,:]

#select 3 beers at random
df_no_top3 = df_no_top3.sample(n = 3)

#the top 6 beers with 3 top popular beers and 3 randomly selected beers
top6 = list(top3['beer'])+ list(df_no_top3['beer'])
```

Figure 5: Recommending by neighbours' neighbour

3.5 Generating Recommendation List

Now we combine all the above functions together to generate a list of recommended beers. Since we used beer ID in our datasets, it would be much more user-friendly if the output is a list of beer names instead of a string of numbers. To achieve this, we make a beer ID to beer name mapping from `gb_beers.csv` and store it as a dataframe. This list is then passed to the `recomend` function where we find beers to recommend by its ID using several functions, followed by mapping each ID to its corresponding name, before returning the final list of beer names. Key code for this is shown in Fig.6.

```
def recommend (usr_id, usr_level, g, edges, usr_usr, pop, beer_list):

    com_rev = pd.read_csv("Data/gb_beer_reviews.csv")
    hub_rev = pd.read_csv("Data/gb_beer_reviews.csv")
    nn_rev = pd.read_csv("Data/gb_beer_reviews.csv")

    usr_type = profile(usr_id,usr_level)
    beers = []
    hub_rec = random.sample(hub(hub_rev, g),5)
    for x in hub_rec:
            beers.append(beer_list.loc[beer_list['beer'] == x, 'name'].iloc[0])

    # if user selected is a 'heavy' user, use community algorithm
    if usr_type == 1:
        com = random.sample(beer_by_com(g, usr_id, edges, com_rev),5)
        for x in com:
            beers.append(beer_list.loc[beer_list['beer'] == x, 'name'].iloc[0])

    # if user selected is an 'occasional' user, use alter's alter algorithm
    else:
        alter = random.sample(NN(usr_id,usr_usr,pop,nn_rev),5)
        for x in alter:
            beers.append(beer_list.loc[beer_list['beer'] == x, 'name'].iloc[0])
    return beers
```

Figure 6: Generating recommendation list

3.6 Results

Fig.7 is an example of the expected result of our recommender system. It is straightforward to see that when we input a user ID (in this case is 988388), and call the recommender function, then we obtain our desired outcome - a list of beers to recommend to this particular user.

```
1  usr_id = '988388'
```
executed in 4ms, finished 02:21:14 2018-12-13

```
1  recommender = recommend(usr_id, usr_level, g, edges, usr_usr, pop, beer_list)
```
executed in 2m 6s, finished 02:23:21 2018-12-13

```
1  recommender
```
executed in 21ms, finished 02:23:21 2018-12-13

```
['Fourpure / Devils Peak - Coastline',
 'Sourdough',
 'DDH Pale Citra Ekuanot Mosaic',
 'Fred in London',
 'Magic Rock / Verdant - What Are The Odds ?',
 'Innis & Gunn India Pale Ale',
 'Oatmeal Stout',
 'Foundation Bitter',
 'Far Skyline',
 'Collingwood']
```

Figure 7: An example of the result

4 Limitation and Future Work

The first limitation of this project would be the fact that our network is established solely based on the data of user interactions on the forum. It does not take users who have rated a lot of beers, but has few or no interaction into account. If we had the user-rating network, our recommender system would be more comprehensive and complete.

Secondly, since our datasets are so huge that it is practically impossible to use the Girvan-Newman algorithm for community detection and betweenness centrality for hubs identification, our solution to this issue is to use an alternative way using functions in `igraph`. Admittedly it help us to achieve the desired purpose (so that we can process to the next step) a lot more quickly than the traditional way in `NetworkX`. However, we are unfamiliar with the theory behind it, so it would be less rigorous than the traditional methods. A question would be whether an upgrade of CPU can help speeding up the Girvan-Newman algorithm.

In the future, we are intrigued to explore other types of recommender system, such as Content-Based Filtering system, provided that we have more information about the target product itself.

Moreover, we could run our recommender system repeatedly and obtain a large amount of recommendation lists. Then we could visualise them using techniques we acquired from Data Visualisation module, or investigate if there is any valuable statistical information that worth further analysis.

Last but not least, it is currently quite difficult for us to testify how accurate or inaccurate are the recommendations. If we could add some interactive features, such as a “not interested” or “I want to try it later” button to each beer, then collect and analyse these data, it is likely that we could greatly improve the accuracy of our recommendation.

Bibliography

Alzahrani, T. & Horadam, K. J. (2016), Community detection in bipartite networks: Algorithms and case studies, *in* 'Complex Systems and Networks', Springer, pp. 25–50.

Barabási, A.-L. et al. (2016), *Network science*, Cambridge university press.

Fish, T. (2009), *My Digital Footprint A two-sided digital business model where your privacy will be someone else's business!*, futuretext.

Girvan, M. & Newman, M. E. (2002), 'Community structure in social and biological networks', *Proceedings of the national academy of sciences* **99**(12), 7821–7826.

Karypis, G. (2001), Evaluation of item-based top-n recommendation algorithms, *in* 'Proceedings of the tenth international conference on Information and knowledge management', ACM, pp. 247–254.

Lops, P., De Gemmis, M. & Semeraro, G. (2011), Content-based recommender systems: State of the art and trends, *in* 'Recommender systems handbook', Springer, pp. 73–105.

McPherson, M., Smith-Lovin, L. & Cook, J. M. (2001), 'Birds of a feather: Homophily in social networks', *Annual review of sociology* **27**(1), 415–444.

Pons, P. & Latapy, M. (2005), Computing communities in large networks using random walks, *in* 'International symposium on computer and information sciences', Springer, pp. 284–293.

Resnick, P. & Varian, H. R. (1997), 'Recommender systems', *Communications of the ACM* **40**(3), 56–58.

Rob Parkin (2004), 'Identifying influencers with social network analysis'.
URL: *https://www.pulsarplatform.com/blog/2014/identifying-influencers-with-social-network-analysis*

Sarwar, B., Karypis, G., Konstan, J. & Riedl, J. (2001), Item-based collaborative filtering recommendation algorithms, *in* 'Proceedings of the 10th international conference on World Wide Web', ACM, pp. 285–295.

Sun, Z., Han, L., Huang, W., Wang, X., Zeng, X., Wang, M. & Yan, H. (2015), 'Recommender systems based on social networks', *Journal of Systems and Software* **99**, 109–119.

Watts, D. J. (2004), *Six degrees: The science of a connected age*, WW Norton & Company.