Sapienza Università di Roma

Data Mining Technology for Business and Society

Homework 2

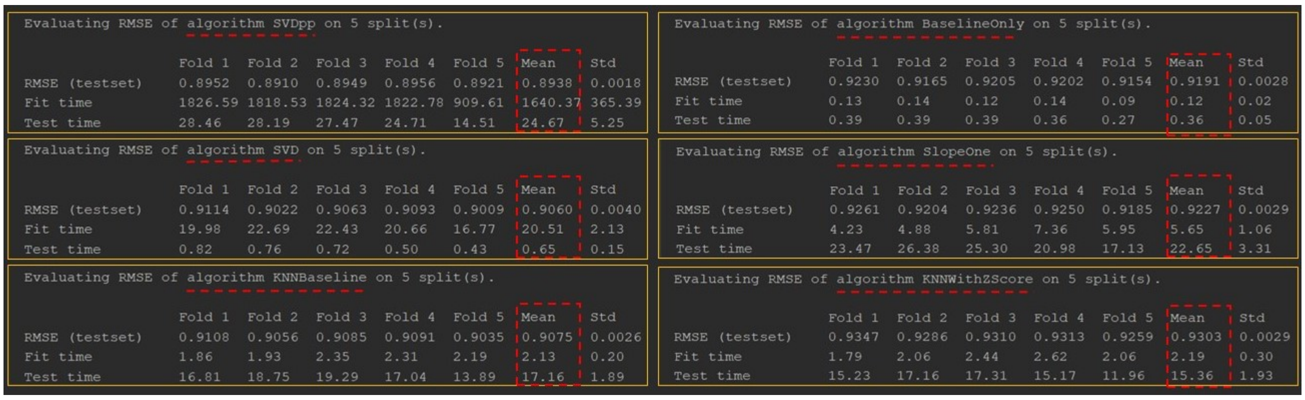
April 2020

1 Recommendation-System

1.1 Recommendation system algorithms

For this first part of the HW, all 11 Recommendation Algorithms, set with default configurations and provided by the Surprise library (Prediction Algorithms), have been applied on the ”ratings.csv” file (structured as follows: user, item and rating). For Every prediction Algorithm, 5-Fold Cross Validation has been computed. The Surprise library offers different kind of error measures, and for the purpose of this task it has been decided to use RMSE bacause it adds normalization and the error is on the same scale of the input data.

Because Cross Validation metodology and moreover Hyperparamiters tuning require a lot of effort, it is suggested to use all CPU-cores. In order to exploit all CPU-cores available on the used machine the ’n_jobs’ parameter (for Cross Validation, GridSEarchCV, RandomizedSearchCV) has been customized by setting it with the value equal to 4.
The tables below show the results returned by the Cross Validation method per each Algorithm. The Algorithms are returned in a descending order, from the best one to the worst one.



```
Evaluating RMSE of algorithm SVDpp on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8952  0.8910  0.8949  0.8956  0.8921  0.8938  0.0018
Fit time          1826.59 1818.53 1824.32 1822.78 909.61  1640.37 365.39
Test time         28.46   28.19   27.47   24.71   14.51   24.67   5.25
```

```
Evaluating RMSE of algorithm BaselineOnly on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9230  0.9165  0.9205  0.9202  0.9154  0.9191  0.0028
Fit time          0.13    0.14    0.12    0.14    0.09    0.12    0.02
Test time         0.39    0.39    0.39    0.36    0.27    0.36    0.05
```

```
Evaluating RMSE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9114  0.9022  0.9063  0.9093  0.9009  0.9060  0.0040
Fit time          19.98   22.69   22.43   20.66   16.77   20.51   2.13
Test time         0.82    0.76    0.72    0.50    0.43    0.65    0.15
```

```
Evaluating RMSE of algorithm SlopeOne on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9261  0.9204  0.9236  0.9250  0.9185  0.9227  0.0029
Fit time          4.23    4.88    5.81    7.36    5.95    5.65    1.06
Test time         23.47   26.38   25.30   20.98   17.13   22.65   3.31
```

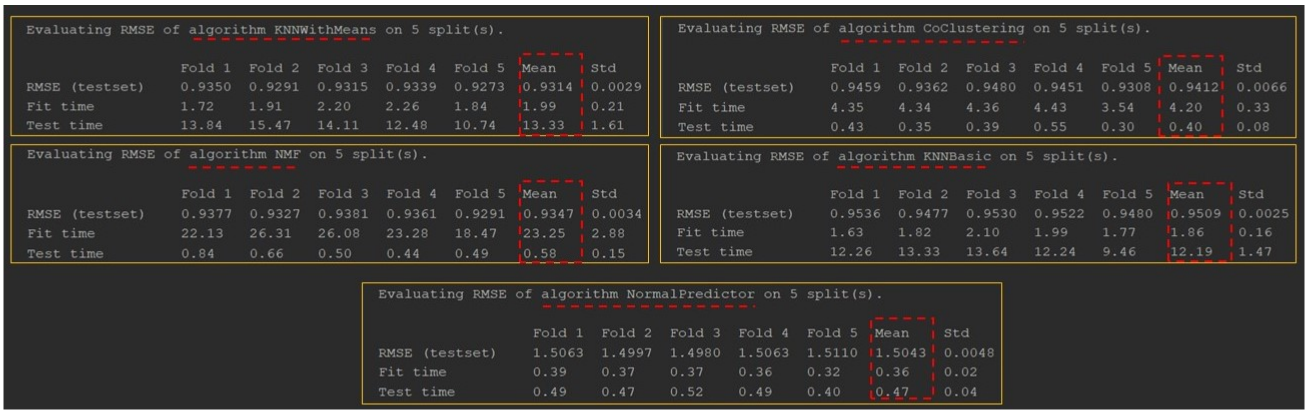
```
Evaluating RMSE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9108  0.9056  0.9085  0.9091  0.9035  0.9075  0.0026
Fit time          1.86    1.93    2.35    2.31    2.19    2.13    0.20
Test time         16.81   18.75   19.29   17.04   13.89   17.16   1.89
```

```
Evaluating RMSE of algorithm KNNWithZScore on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9347  0.9286  0.9310  0.9313  0.9259  0.9303  0.0029
Fit time          1.79    2.06    2.44    2.62    2.06    2.19    0.30
Test time         15.23   17.16   17.31   15.17   11.96   15.36   1.93
```

(a) First six Recommendation Algorithms



```
Evaluating RMSE of algorithm KNNWithMeans on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9350  0.9291  0.9315  0.9339  0.9273  0.9314  0.0029
Fit time          1.72    1.91    2.20    2.26    1.84    1.99    0.21
Test time         13.84   15.47   14.11   12.48   10.74   13.33   1.61
```

```
Evaluating RMSE of algorithm CoClustering on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9459  0.9362  0.9480  0.9451  0.9308  0.9412  0.0066
Fit time          4.35    4.34    4.36    4.43    3.54    4.20    0.33
Test time         0.43    0.35    0.39    0.55    0.30    0.40    0.08
```

```
Evaluating RMSE of algorithm NMF on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9377  0.9327  0.9381  0.9361  0.9291  0.9347  0.0034
Fit time          22.13   26.31   26.08   23.28   18.47   23.25   2.88
Test time         0.84    0.66    0.50    0.44    0.49    0.58    0.15
```

```
Evaluating RMSE of algorithm KNNBasic on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9536  0.9477  0.9530  0.9522  0.9480  0.9509  0.0025
Fit time          1.63    1.82    2.10    1.99    1.77    1.86    0.16
Test time         12.26   13.33   13.64   12.24   9.46    12.19   1.47
```

```
Evaluating RMSE of algorithm NormalPredictor on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.5063  1.4997  1.4980  1.5063  1.5110  1.5043  0.0048
Fit time          0.39    0.37    0.37    0.36    0.32    0.36    0.02
Test time         0.49    0.47    0.52    0.49    0.40    0.47    0.04
```

(b) Last five Recommendation Algorithms

Recommendation Algorithm	Mean RMSE
SVDpp	0.8938
SVD	0.9060
KNNBaseline	0.9075
BaselineOnly	0.9191
SlopeOne	0.9227
KNNWithZScore	0.9303
KNNWithMeans	0.9314
NMF	0.9347
CoClustering	0.9412
KNNBasic	0.9509
NormalPredictor	1.5043

(c) Ordered Recommendation Algorithms by their mean RMSE

Figure 1: Recommendation Algorithms Results

1.2 Hyperparamiter Tuning

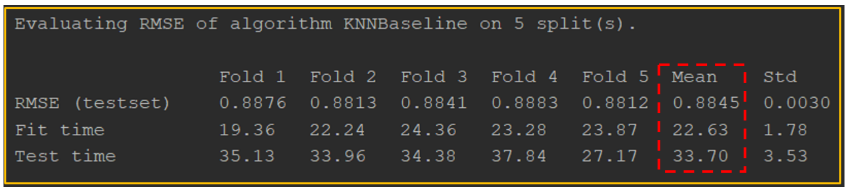
In order to improve the quality of KNNBaseline and SVD Recommendation Algorithms, RandomSearchCV and GridSearchCV have been performed in order to tune the hyper-parameters and find a good configuration that allows to improve the errors. An acceptable configuration has to return an average RMSE over all five folds (beacuse 5 fold cross validation has been used) less than 0.89. Tuning hyperparamiters is a compromise between quality output and time. Trying many combinations increases the probability to find a very good combination but, on the other hand, more combinations implies a greater computational time.

1.2.1 Hyperparamiter Tuning KNNBaseline Recommendation Algorithm paramiters by using RandomSearchCV

Because RandomSearchCV doesn't use an exhaustive combinatorial approach, for every parameter a basket of possible values is defined, and the RandomSearchCV picks randomly samples from this parameter space. For that reason, it is necessary to point out that 60 different combinations have been tried. The time required by the machine to execute this task has been of 63.8 minutes. The hyperparamiter space is described by Table 1, where the last yellow column shows the chosen configuration of every parameter at the end of the RandomSearchCV Algorithm. Whereas Figure 2 shows the 5 FoldCV results by using the previously discovered parameters configuration.

model	**options**	**parameter**	**tested values**	**chosen value**
KNNBaseline		k	[n for n in range(10,100,10)]	40
KNNBaseline		min_k	[n for n in range(1,40,2)]	11
KNNBaseline	sim_options	name	[cosine, msd, pearson, pearson_baseline]	'pearson_baseline'
KNNBaseline	sim_options	user_based	[True, False]	False
KNNBaseline	sim_options	min_support	[n for n in range(1,10,1)]	4
KNNBaseline	bsl_options	method	sgd	sgd
KNNBaseline	bsl_options	n_epochs	[n for n in range(20,80,10)]	50
KNNBaseline	bsl_options	reg	[n for n in np.arange(0.02,0.1,0.01)]	0.08
KNNBaseline	bsl_options	learning_rate	[n for n in range(0.002,0.01,0.001)]	0.004

Table 1: KNNBaseline tested and chosen parameters values


```
Evaluating RMSE of algorithm KNNBaseline on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8876  0.8813  0.8841  0.8883  0.8812  0.8845  0.0030
Fit time          19.36   22.24   24.36   23.28   23.87   22.63   1.78
Test time         35.13   33.96   34.38   37.84   27.17   33.70   3.53
```

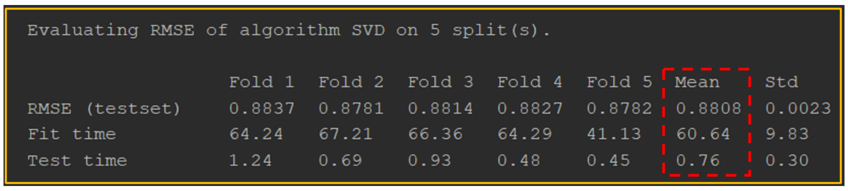
Figure 2: 5_FoldCV for KNNBaseline by using best parameters

1.2.2 Hyperparameter Tuning SVD Recommendation Algorithm parameters by using GridSearchCV

Because GridSearchCV uses an exhaustive combinatorial approach, the parameter space cannot be too wide otherwise it'll take too much time to return the best parameters configuration between the all possible given options. For that reason, 108 combinations have been tried. The time required by the machine to execute this task has been of 154.8 minutes. The hyperparameter space is described by Table 2, where the last yellow column shows the chosen configuration of every parameter at the end of the GridSearchCV Algorithm. Whereas Figure 3 shows the 5 FoldCV results by using the previously discovered parameters configuration.

model	**parameter**	**tested values**	**chosen value**
SVD	n_factors	[n for n in range (50,70,10)]	60
SVD	n_epochs	[n for n in range (60,80,10)]	70
SVD	init_mean	0	0
SVD	reg_all	[0.04, 0.09]	0.09
SVD	lr_bu	[0.0009, 0.002, 0.003]	0.0009
SVD	lr_bi	[0.001, 0.002, 0.003]	0.002
SVD	lr_pu	[0.0009, 0.002, 0.005]	0.005
SVD	lr_qi	[0.0009,0.002,0.009]	0.009

Table 2: SVD tested and chosen parameters values



```
Evaluating RMSE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8837  0.8781  0.8814  0.8827  0.8782  0.8808  0.0023
Fit time          64.24   67.21   66.36   64.29   41.13   60.64   9.83
Test time         1.24    0.69    0.93    0.48    0.45    0.76    0.30
```

Figure 3: 5_FoldCV for SVD by using best parameters

2 Discover Social Communities

This second part of the HW consists on finding communities around the below reported characters in each of the 4 books composing the fantasy novel called by "A Song of Ice and Fire". The 4 characters are:

- Daenerys-Targaryen
- Jon-Snow
- Samwell-Tarly
- Tyrion-Lannister

The interactions between characters are stored inside 4 ".tsv" files called: book_1 (A Game of Thrones), book_2 (A Clash of Kings), book_3 (A Storm of Swards), book_4 (it is the combination of the two books: Feast for Crows & A Dance with Dragons). Every row of every ".tsv" file represents the fact that the names of the two characters corresponding to the first and second column appeared within 15 words of one another in the corresponding book. In order to accomplish the request, for each book of the series, an unweighted and undirected graph, where nodes refer to the characters and edges the interactions among them, must be build up.
A community in a graph, from a qualitative prospective, represents a set of nodes that are more connected between them than with the rest of the graph. The quality of a community is measured by the Conductance, the lower it is the higher the community quality is.

$$Conductance(S) = \frac{links(S, V\text{-}S)}{links(S, V), links(V\text{-}S, V)}$$

Because computing the conductance for each possible combination of nodes is almost impossible for big graphs, another approach, based on PageRank, is used.
In order to find good local-community it is necessary to compute the Personalized-PageRank (PPR), that is a specification of the the more general Topic-Specific-PageRank, for each node in the graph. The computation of PageRank depends on the so called damping factor (α) that rules the Algorithm. Once all the PPR are computed per each node, they are normalized by a factor of Degree(V)$^{\text{exponent}}$. Once all Normalized_scores are computed they are sorted in a descending order and the SWEEP is computed in order to return the set of nodes with the minimum conductance.
For this HW 19 different (α) and 6 different "exponent" values have been provided, so it is necessary to seek out between 114 possible combinations of these 2 factors the best combination that returns the lowest conductance.

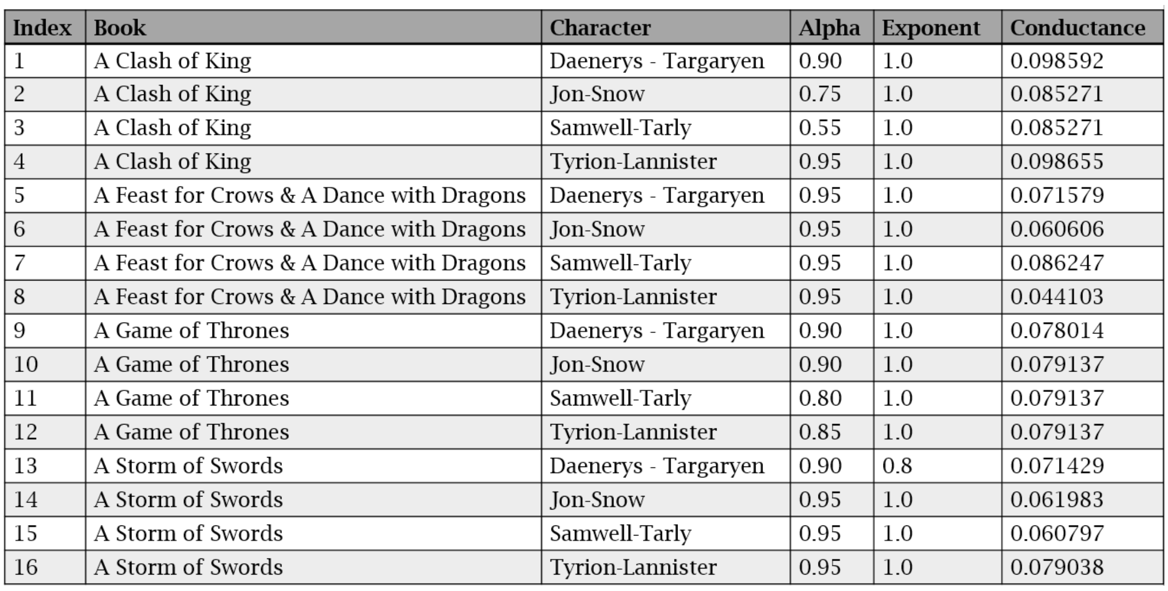
HW2_SW_01_Part2.py contains the code used to accomplish the HW. The code is composed by 4 functions that have been used. The "create_graph()" function is used to build up the undirected graphs.
The "find_community()" function instead returns, for any given Normalized_score list, the best community (i.e. the set of nodes with the lowest conductance). The "best_community()" function, on the other hand, is the core function. It takes in input the graph and the character_name and it iterates over each value of the dumping factor(α) and "exponent". At the end it gives back the community (set of nodes) with the lowest conductance for the given input character. Specifically, the function starts iterating over the set of all possible α-values [n for n in np.arange(0.05, 1,0.05)]. It picks the first and computes the PPR. After that, it starts iterating over all possible "exponent" values [n for n in np.arange(0,1.2,0.2)]. It picks the first one and computes the Normalized_score for each PPR. At the end of this process it returns the Normalized_score list that will be the input for the "find_community()" function, that is called at this point of the Algorithm. Once the "find_community()" returns the best set of nodes, according to the damping factor (α) and the chosen "exponent", it checks if the given community is better than the previous one by checking the conductance values(the algorithm is initialized with a conductance value equal to +infinity). This process is repeated for every possible value of the exponent factor. So, at the end of that loop, it will return the best set of nodes according to the "exponent" and the previously chosen dumping factor.
However, this process is repeated for all possible values that the dumping factor can have. So, at the end of this higher level loop, it will return the best combination of the damping factor (α) and "exponent" that allows the restitution of the set of nodes (i.e. the community) with the lowest conductance. It is necessary to point out that there is not a unique best combination of the two parameters, in fact, some combinations return the same

result. For that reason, if 2 or more combinations of dumping factor (α) and ”exponent” return the same result, only the one that occurs first will be retrieved as the best one. For example, if the lowest conductance, for a given character, is returned by the combination of α= 0.8 & exponent = 0.2 but also from α= 0.9 & exponent = 0.8 than, only α= 0.8 & exponent = 0.2 will be considered because it occurs before. It is also necessary to highlight that conductance equal to 0 or 1 has not been considered as valid, for that reason, if they occur they will be discarded.
The last function is the ”family_counter()” function. It uses the community of every character to count how many characters inside the community belong to Baratheon, Lannister, Stark, Targaryen families.
At the end of everything, 16 communities are returned because there are 4 input characters and 4 books.

The Tables 3 and 4 (sorted by ascending values of the first column and then by ascending values of the second column) below contain the results:

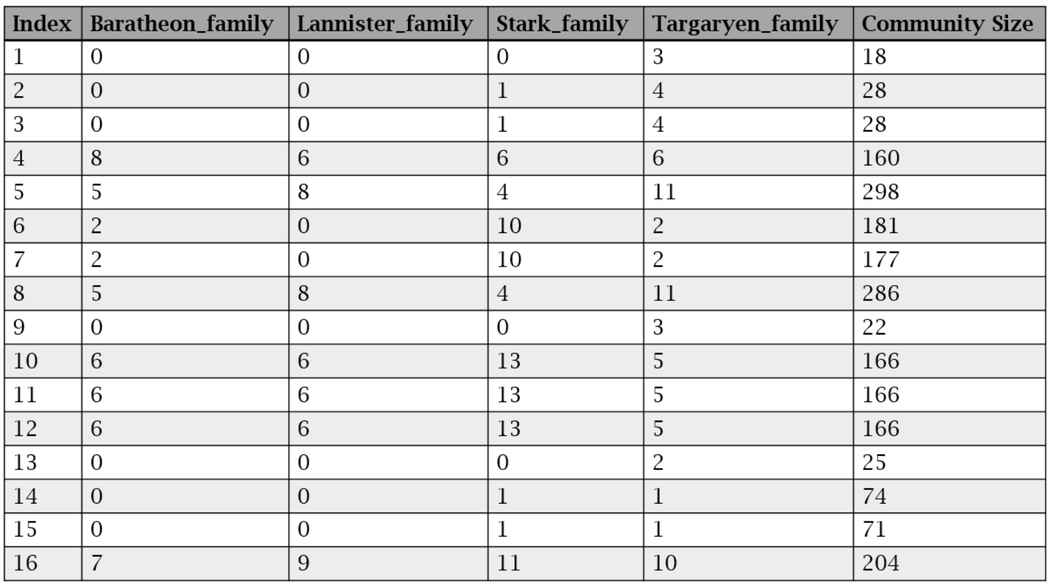
- Table 3 shows: the book novel, the character around which the community has been discovered, the α-value and the ”exponent” that retrieved the lowest conductance (that is shown in the last column).
- Table 4 instead has: 4 columns dedicated to every family (Baratheon, Lannister, Stark, Targaryen) and where every value represents how many nodes in that community (represented by the row) belong to the family. Whereas the last column says the size of every community.

The index column, shown in both tables, is used as reference.



Index	Book	Character	Alpha	Exponent	Conductance
1	A Clash of King	Daenerys - Targaryen	0.90	1.0	0.098592
2	A Clash of King	Jon-Snow	0.75	1.0	0.085271
3	A Clash of King	Samwell-Tarly	0.55	1.0	0.085271
4	A Clash of King	Tyrion-Lannister	0.95	1.0	0.098655
5	A Feast for Crows & A Dance with Dragons	Daenerys - Targaryen	0.95	1.0	0.071579
6	A Feast for Crows & A Dance with Dragons	Jon-Snow	0.95	1.0	0.060606
7	A Feast for Crows & A Dance with Dragons	Samwell-Tarly	0.95	1.0	0.086247
8	A Feast for Crows & A Dance with Dragons	Tyrion-Lannister	0.95	1.0	0.044103
9	A Game of Thrones	Daenerys - Targaryen	0.90	1.0	0.078014
10	A Game of Thrones	Jon-Snow	0.90	1.0	0.079137
11	A Game of Thrones	Samwell-Tarly	0.80	1.0	0.079137
12	A Game of Thrones	Tyrion-Lannister	0.85	1.0	0.079137
13	A Storm of Swords	Daenerys - Targaryen	0.90	0.8	0.071429
14	A Storm of Swords	Jon-Snow	0.95	1.0	0.061983
15	A Storm of Swords	Samwell-Tarly	0.95	1.0	0.060797
16	A Storm of Swords	Tyrion-Lannister	0.95	1.0	0.079038

(a) Table 3: Every row represents a community



Index	Baratheon_family	Lannister_family	Stark_family	Targaryen_family	Community Size
1	0	0	0	3	18
2	0	0	1	4	28
3	0	0	1	4	28
4	8	6	6	6	160
5	5	8	4	11	298
6	2	0	10	2	181
7	2	0	10	2	177
8	5	8	4	11	286
9	0	0	0	3	22
10	6	6	13	5	166
11	6	6	13	5	166
12	6	6	13	5	166
13	0	0	0	2	25
14	0	0	1	1	74
15	0	0	1	1	71
16	7	9	11	10	204

(b) Table 4: Every row represents a community