# Report TX

Benjamin Loriot - GI04

*Suiveur :* Mr. Yves Grandvalet

Printemps 2019

## 1 Introduction

This TX aims to contribute to a project which goal is to aid decision regarding the choice of external open source libraries. The decision should rely on whether the project is well maintained by its contributors, if it contains clean code, if the project is stable (likely to be improved), secure and many other different relevant aspects[1]. The project consists in exploring a large set of open source repositories extracted from GitHub in order to compute relevant metrics. The resulting data, would be stored and made available for further analysis.

For this TX I chose to carry out an analysis focused on developers of open-source projects from GitHub with a focus on Java projects. Inspired by the work of *Montandon et al.* [2], the goal is to assess skills of developers based on the tools they use. I also saw in *Palazzi et al.* that projects evolve into internally organised blocks[3] of contributors. I decided to focus on the libraries imported in the files that a given developer has modified in the past to create such clusters. The main motivation is that if someone modifies a file that requires a library he is more likely to have knowledge in it.

The main difficulty for this project is that I don't have access to labelled data. Thus, I decided to first carry this analysis on a project in which I already have insights. We chose *Spoon*[4], a medium size project being developed since several years by what could be considered to be an average to small community.

## 2 Setup

### 2.1 Data collection

First I extracted commit information from the git repository of the project. In order to do so I 1) cloned the repository 2) checked out all the commits from the master branch 3) extracted the modified files $f_i$, there associated package $P_{f_i}$, and imports $I_{f_i}$. I extracted this information *inside the checked-out commit* to be sure to have the exact imports used by the developer during the commit.

Finally I extracted some extra information about the commit such as its *id*, *author* (the one who wrote the code), and *commiter* (the one who committed the changes, merged it). Later this data is stored in a `.json` file.

### 2.2 Embedding of commits

Then I mapped imports into unique ids. In order to reduce the number of different ids I first defined a *cut* function that would cut imports at a given depth $d$ (e.i. `org.junit.Assert` $\rightarrow$ `org.junit`). I made the assumption that depth of $d = 2$ would roughly correspond to unique libraries. Later I created a map function $Id$ that would affect unique ids to these imports.

Finally I was able to represent files as equal to :

$$\vec{f} = \sum_{i \in I'_f} e_i$$

where,

$$I'_f = \{Id(cut(i)) \setminus i \in I_f\}$$

and commits such as,

$$\vec{C} = \sum_{f \in C} \vec{f}$$

.

### 2.3 Authors

If several authors have the same email address they are merged into a single author, this is meant to avoid duplicated authors.

## 3 Data analysis

### 3.1 Summary

I collected data from 2344 commits, developed by 60 contributors, composed of 2473 different imports mapped into 149 elements. Latter I dropped elements imported less than once to finally have 117 import groups.

### 3.2 Commit distribution

The extracted matrix is available in Figure 1.

The matrix seems to be well structured with some more important contributors and libraries.

In order to have a better view at the data I ploted the same matrix but sorted by the most important contributors and libraries, Figure 2. We can clearly see some sort of exponential law appearing.

Even if the matrix is quite homogeneous, not all developers follow the exact same pattern in terms of usage of libraries. For example, the authors *dwayneb* and *noguera* do not seem to be using the imports `org.junit` which can indicate that they did not provide tests for what they have developed [1].

## 4 Normalisation

In this study I considered two different kinds of normalisation, the first one is $L2$ normalisation and a second one aims to accentuate the expertise of a developer.

[1]A finner analysis lead to the conclusion that junit was simply not used in the project at this time.

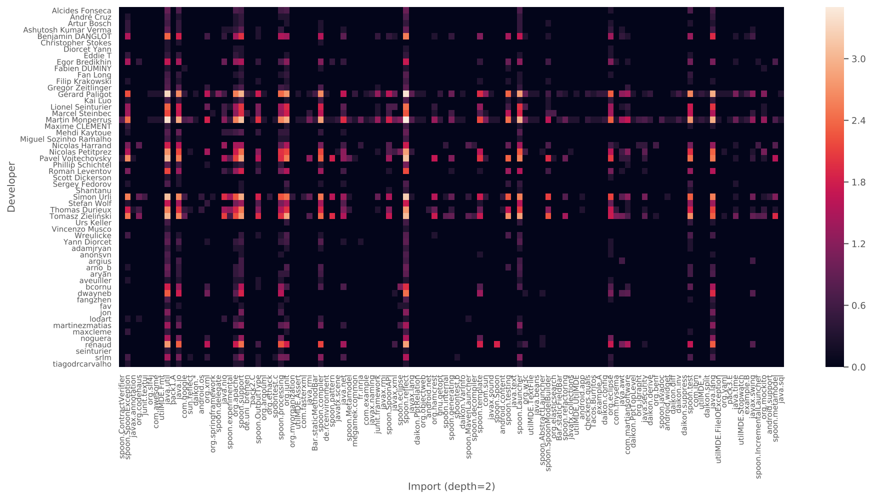

Figure 1: Developer/Import matrix, log scale.

## 4.1 Normalisation L2

### 4.1.1 Definition

For each developer I divided their imports by the norm of the vector. This normalisation aims at representing the relative usage of libraries no matter the total amount of contributions. The new matrix is presented Figure 3.

The two main takeaways of the new normalisation are that :

- developers seem to be following similar patterns. The three most used imports are `spoon.reflect`, `spoon.support` which are core packages from the project, and `java.util`, a key java library.
- if we consider less used libraries we can see some discrepancies, manly on small contributors.

I will now focus on extracting this patterns between developers.

### 4.1.2 PCA

First I realised a PCA. If we take a look at the explained variance, Figure 4, we can notice that the 2 first principal components only represent 40% of the total explained variance. Twenty dimensions out of the 55 are sufficient to represent almost all the explained variance. This indicates that the number of dimensions could be reduced by 3 without loosing information. However, we would lose the link between authors and libraries.

A plot representing the 3 first principal components of the most active contributors is available in Figure 5.

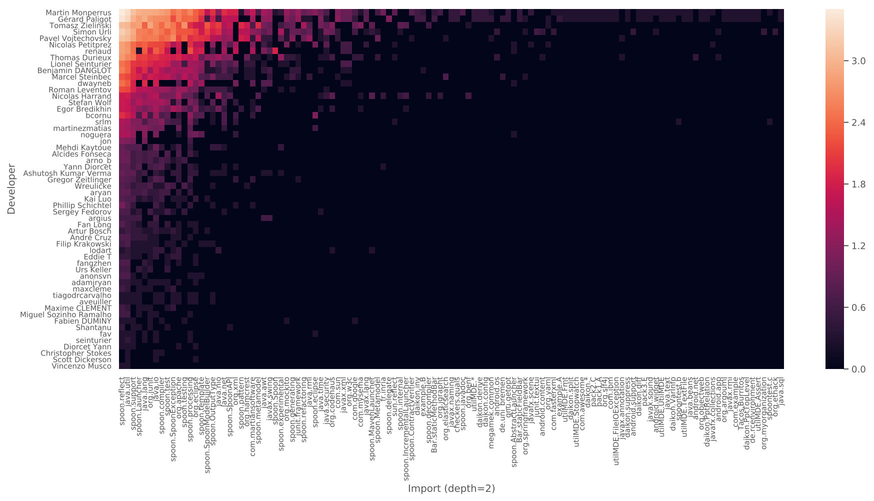

Figure 2: Ordered Developer/Import matrix, log scale.

It is interesting to notice how clusters seem to be related to time. For example the blue cluster is composed of *bcornu*, *dwayneb*, *renaud* and *noguera* who are the orginal contributors of the project. On the other side of the graph we can also see the yellow cluster that contains more recent contributors such as *Nicolas Harrand* or *Tomasz Zielinski*.

Nothing indicates that time would appear from imports. However, if we have a closer look to the contributions of the 20 most important initial variables (the imports) over the 2 first principal components we can notice that there are two libraries `spoon.reflect` and `java.util` that are pushing contributors to the bottom left corner. Whereas, every other variables is pushing contributors to the top right corner of the pca. `org.junit` and `spoon.test` are also opposed to `spoon.reflect` and `java.util`.

One interpretation would be that what mostly separates these two clusters is :

- The complexity of the project, a bigger project will tend to have more divers imports that will balance the importance of the 2 most used imports;
- The presence of tests inside the project.

These two factors can easily be related to time. Complexity of the project increases with time, tests are more common in bigger projects and automated testing has recently gain in popularity with the popularisation of automated CI environments such as *Jenkins* or *TravisCI*

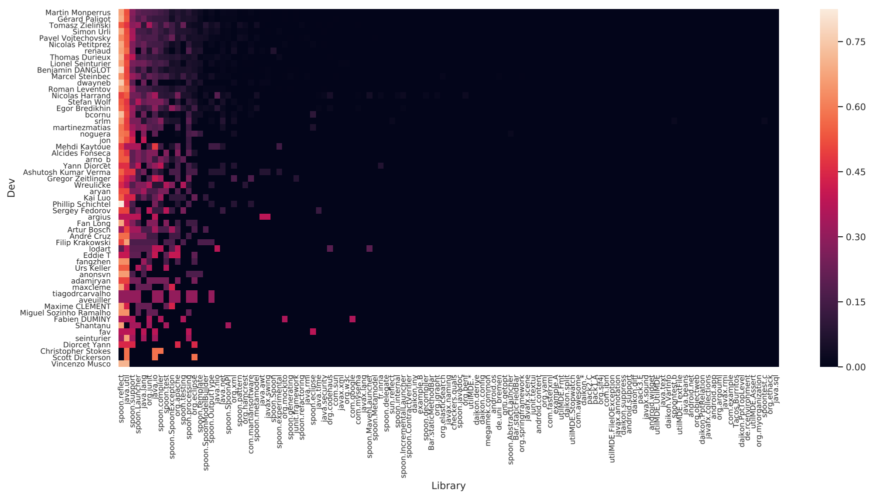

Figure 3: Developer/Import matrix, after L2 normalisation by rows.

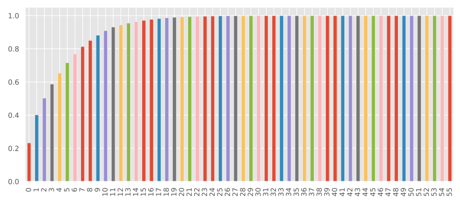

Figure 4: Sum of the explained variance of each principal components of the L2-normalised Developer/Import matrix.

### 4.1.3 Co-Clustering

**Co-Clustering** The goal of Co-Clustering is to rearrange rows and columns in order to obtain clusters. We can identify two kinds of co-clustering algorithms, diagonal algorithms that will rearrange the rows and columns in order to obtain a diagonal, and Non-Diagonal algorithms that are not restricted. Non-Diagonal algorithms can have different numbers of clusters in rows and columns, this makes the selection of the best values for each argument (n_columns n_rows) much harder because the best partition will be as many clusters as you have rows and columns. On the other hand Diagonal algorithm will reach a point where increasing the number of clusters won't have any major effect.

The main interest in using co-clustering algorithms is to exhibit relations between rows and columns and between clusters.

For this study I used the python package *CoClust*[5]. *CoClust* implements two Diagonal algorithms (*CoclustMod* and *CoclustSpecMod*) based on maximisation of the modularity of the graph represented by the matrix. The major difference

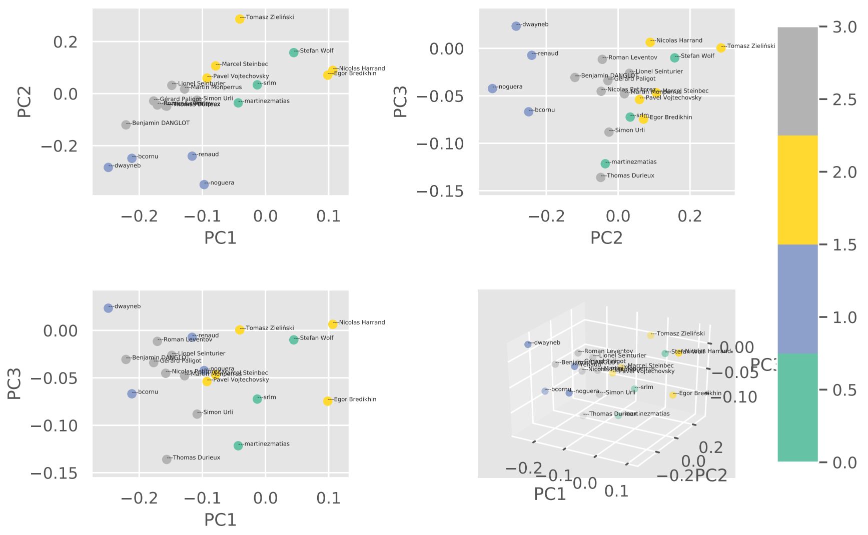

Figure 5: The three firsts principal components of the PCA realised on the L2-normalised Developer/Import matrix.

is that CoclustSpecMod uses a variance of modularity which is called specular modularity, this variance tends to balance more clusters' size.

**Modularity** Modularity is one measure of the structure of networks or graphs. It was designed to measure the strength of division of a network into clusters. Networks with high modularity have dense connections between the nodes within clusters but sparse connections between nodes in different clusters. Modularity Q is defined as the fraction of edges that fall within group $a$ or $b$, minus the expected number of edges within groups $a$ and $b$ for a random graph with the same node degree distribution as the given network.

**Application** The normalised matrix cannot be used for Co-Clustering. In fact, since rows and columns are very unbalanced, few columns are full while the rest is very close to 0. On the other hand rows are very similar.

I decided to run the co-clustering algorithm on the binary matrix $M'$ defined as following

$$M'_{i,j} = \begin{cases} 1 & \text{if } M_{i,j} > 0 \\ 0 & \text{else.} \end{cases}$$

.

To assess the best number of clusters I studied modularity of the partitioned graph for different values of $n$. After $n \geq 3$ the results stay stable around 0.29, then I decided to set the number of clusters equal to 3.

Interestingly, I get similar results than with PCA. The major factor that diferenciates authors is diversity. First we have the cluster of the historical or

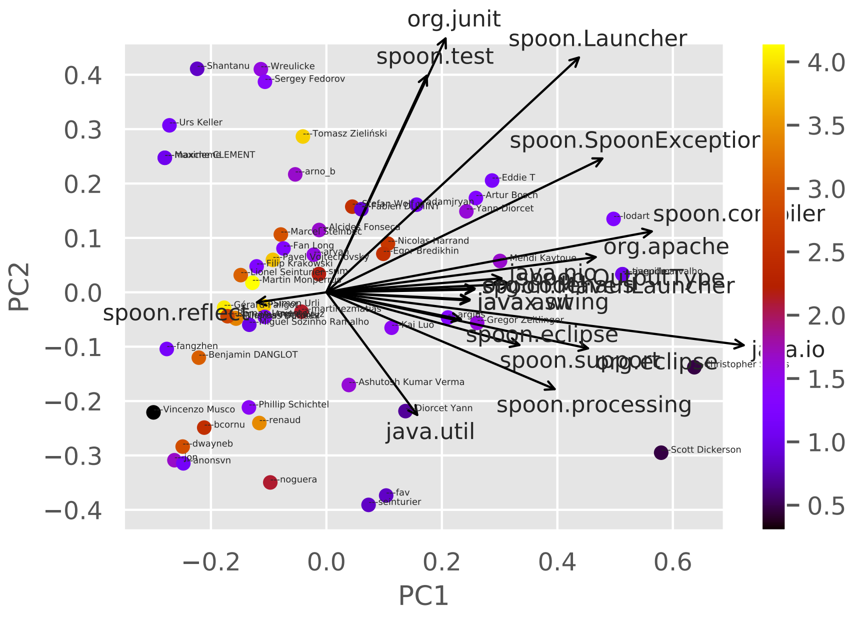

Figure 6: Projection of the top 20 most important axis of the PC-1 and 2 plan. Colour represents the log of the total number of contributions.

small developers who are using basic libraries and then we have the cluster of developers that used more complex libraries. We can see that the second cluster is strongly related with the first one. This can be explained by the fact that even if the second cluster tends to use more divers libraries they are still using the basic ones that are part of the core of the project.

## 4.2 Normalisation 'expertise'

### 4.2.1 Defininition

This normalisation consists in first applying a $L1$ normalisation over the columns (libraries) and then a $max$ normalisation over the rows (authors) in order to have values between 0 and 1. The goal is to find which libraries contributors have used the most relatively to the total usage of libraries and their total number contributions. By doing so we hope to find abnormal usage of a given library by a developer which could indicate some sort of expertise.

However the major issue of this normalisation is that it will tend to overestimate the importance of small libraries used only few times by a single developer. A solution would be to ignore libraries used less than n times.

Here we chose $n = 10$. We obtained final normalised matrix shown in Figure 9.

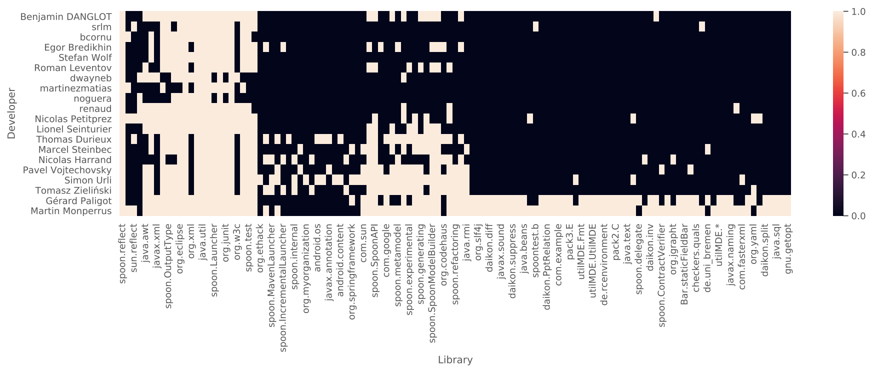
Figure 7: Co-Clustering carried on the matrix M'.

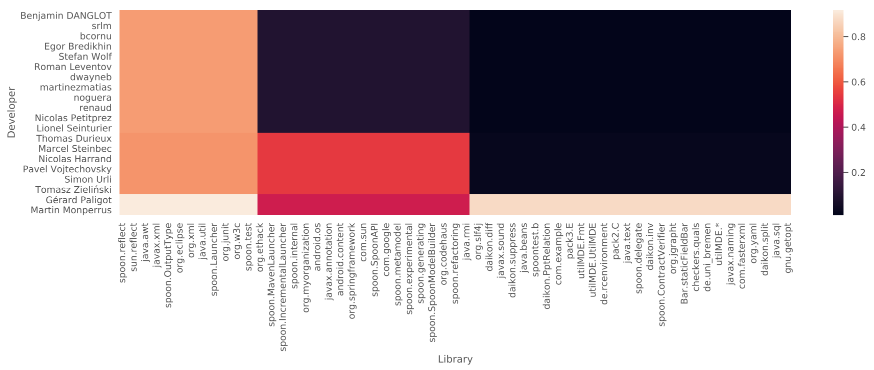
Figure 8: Summary of the Co-Clustering carried on the matrix M'.

### 4.2.2 Co-Clustering

I applied a similar process as the one defined in subsubsection 4.1.3. The time modularity indicates a number of clusters equal to 4. Results are shown in Figure 10, while a schematic view is available in Figure 11.

Interestingly, I got similar clusters as with the k-means in subsubsection 4.1.2 or Co-Clustering (subsubsection 4.1.3) of the L2 normalised matrix. However, the diagonal seems to sharper and imports that discriminate developers from each other are more highlighted .

For example, in both Figure 8 and Figure 11 we can identify the core import cluster, respectively, the 1 column and the 3 column. However, for most of the clusters this row is relatively less important compared to the diagonal. This may be interesting as the most used imports are not useful to identify expertise.

However, I would need more information about imports and developers in order to be able to provide more detailed conclusions.

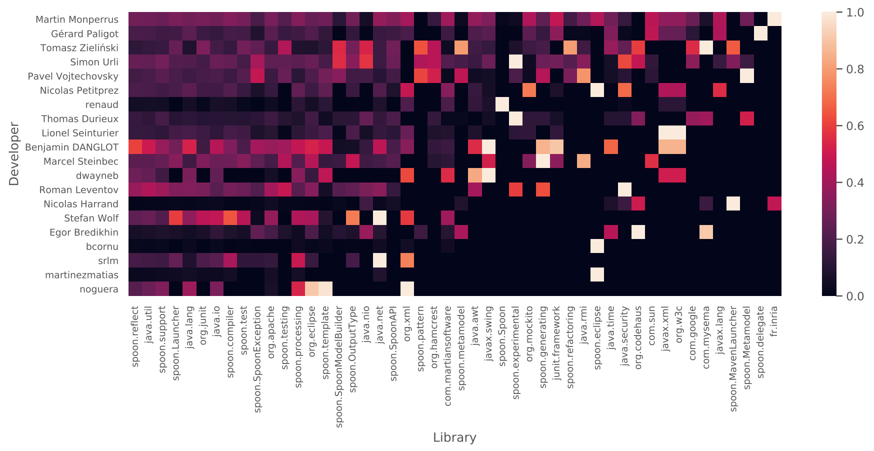

Figure 9: Developer/Import matrix after 'expertise' normalisation and suppression of the least important libraries and developers.

## 5 Possible Improvements

### 5.1 Take into account the relations between imports

For the moment I don't take the hierarchical relations between imports into account. For example, both `android.content` and `android.net` are from the global package `android` while `java.lang` is from Java core. I could use the tree structure of Java import to create a forest of libraries. Then I could compute the Laplacian of the graph and the Symmetric normalised Laplacian which is a symmetrical, positive definite matrix. This matrix could be used as a new definition of scalar product that would take into account relations between libraries.

### 5.2 Define a better mapping that is closer to library

Here I made the assumption that, in Java, mapping imports at a depth of 2, as defined in subsection 2.2, would correspond to unique libraries. However, this statement represents an important approximation of reality. In reality developers are free to chose the name of their package.

### 5.3 Attach labels to imports in order have more information on their purpose

Once I have defined a better mapping of imports to libraries I would be able to retrieve more information about the imports. For example, I could extract more general information on maven central. With this information in mind we may be able to define more complex relations between libraries.

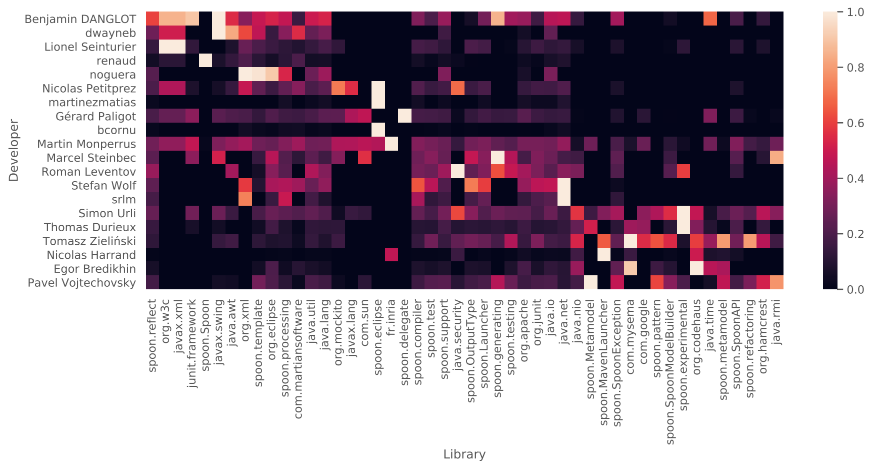

Figure 10: Co-Clustering carried on the 'expertise' matrix.

## 6 Conclusion

In this TX project I had the opportunity to study the relation between developers and the libraries they import in their code. We saw that patterns can be highlighted more or less sharply depending on the data normalisation we choose to use.

However, this patterns are also strongly related to the current state of the project itself, and current trends. In order to be able to provide a deeper analysis we would need to extract more information regarding relations between the libraries and purpose.

Work still needs to be carried out on this project.

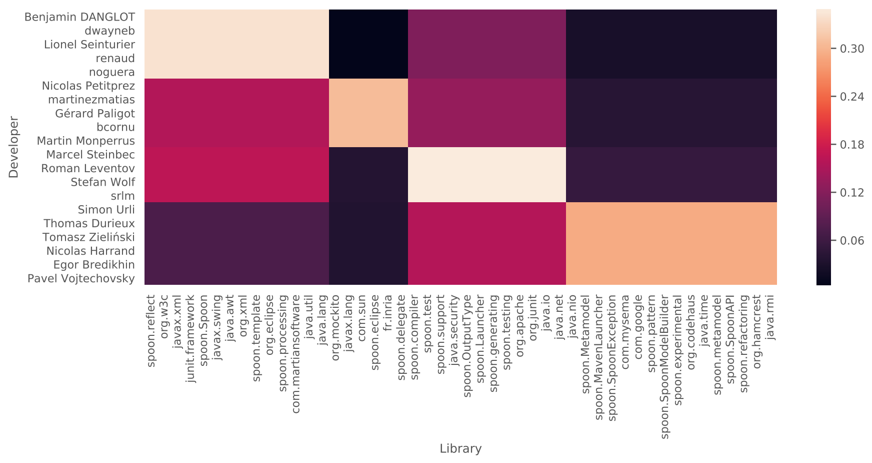

Figure 11: Summary of the Co-Clustering carried on the 'expertise' matrix.

## References

[1] S. Jansen, "Measuring the health of open source software ecosystems: Beyond the scope of project health," *Information Software Technology*, vol. 56, pp. 1508–1519, 2014.

[2] J. E. Montandon, L. L. Silva, and M. T. Valente, "Identifying experts in software libraries and frameworks among github users," *CoRR*, vol. abs/1903.08113, 2019. [Online]. Available: http://arxiv.org/abs/1903.08113

[3] M. J. Palazzi, J. Cabot, J. L. Cánovas Izquierdo, A. Solé-Ribalta, and J. Borge-Holthoefer, "Online division of labour: emergent structures in Open Source Software," *arXiv e-prints*, p. arXiv:1903.03375, Mar 2019.

[4] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[5] F. Role, S. Morbieu, and M. Nadif, "Coclust: A python package for co-clustering," *Journal of Statistical Software, Articles*, vol. 88, no. 7, pp. 1–29, 2019. [Online]. Available: https://www.jstatsoft.org/v088/i07

## Appendix

In subsubsection 4.1.3 I mentioned that the Co-Clustering algorithm I used is based on graphs. We can compare the results with the graph represented by the 'expertise' normalised adjacency matrix defined subsubsection 4.2.1. If we compare results with Figure 11 we can see similar results.

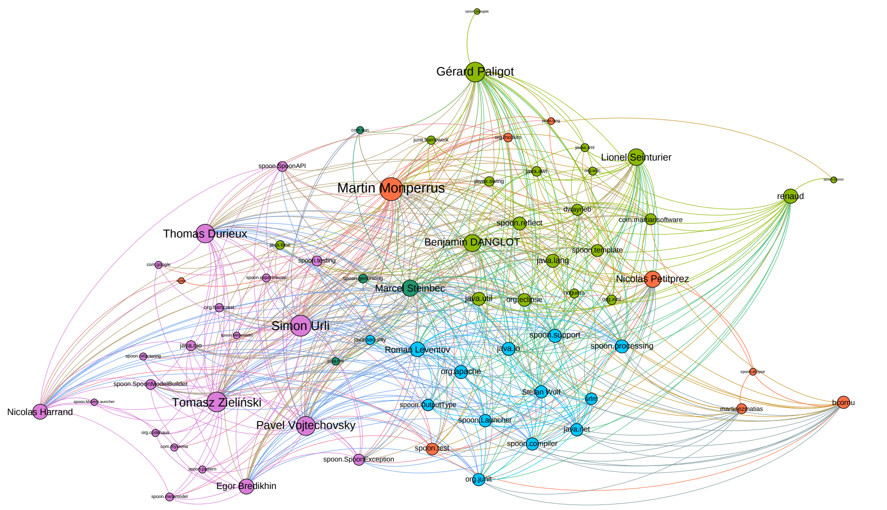

Figure 12: Representation of the 'expertise' graph. For specialisation we used *Force Atlas 2* algorithm and a modularity based approach for clustering.