CodeMips – documentation

Submitted by Uriel Hai & Mendi Halpern

Our goal

In consultation with our advisor (Dr. Amir) we wanted to adapt the MIP-Net model to the domain of group collaboration in a programing project.

Specifically, we wanted to test the feasibility of using the MIP-Net model in order to help programmers working in groups to get a personalized update-order on changes made by others to the code, so that the changes that are more important for a given user will be shown to him\her first.

For this purpose, we focused on Github repositories and used the model to predict which files a given user will modify in his\her next commit.

“future modification” is used as proxy to estimate in which files a given user is interested, and thus is important for him\her to receive updates on changes made by others in these files.

Model description

The concept of MIP-Net is explained in detail in Dr. Amir research paper, which is the basis to this project, as mentioned before.  
In this project, we used a (slightly modified) version of the MIP-Net to represent the setting of a team working on a programing project.

A “session” is an update to the programing-project by the user. We used Github’s “commit” as a “session”. An “action” is interaction with a file (can be add, remove, rename, delete).

In the graph, object-nodes are files in the project, and user-nodes are contributors.

The weights on the edges represent dynamic “connection” between the vertexes. Short description: Every time a user interacted with a code-file, we add weight to the edge between their nodes (u-ao edge), and the same for two objects which are interacted with in the same session (ao-ao edge). When only one of the nodes of an edge appear in a session, a decay factor is activated for that edge.

When a user asks for a ranking (in our simulations: at the beginning of each session, or before every commit), every object’s node is ranked by Degree-Of-Interest to the user and returned by this order.

DOI(user,ao) is computed by the following formula:

.

Project modules (in a nutshell)

**CSR.py** - Translating commits into sessions for emitting to the MIP graph

**Entities.py** – Implementing “Session” and “Action” class for the MIP

**MIP.py** - Full Implementing of the MIP-Net model  
  
**DataModule** – classes for retrieving, cleaning and structuring repositories from Github.  
  
**AnalysisModule** – Scripts for running the MIP model on Github data, visualizing and analyzing the results.

Differentiation for original MIP-graph description

**Similarity metric**:

We’ve identified a potential problem with the adamic-Adar-Proximity function in the case where the two nodes (s,t) are connected by a third node (c) s.t. total weight of the edges (c,s),(t,c) is 1. in that case, the formula should return inf (0 in the denominator).

We decided to use the “simple” Proximity metric instead.  
  
**Handling deleted objects:**  
Deleted objects are kept in the graph until the weight of all connected edges is 0. This allow recovery in cases were one commit deletes a file and another quickly re-instate it, maintaining the “information” stored in the object (derived from past activity) about connections with other nodes.

The only pitfall is in the case where a file is deleted, and shortly after, a new file is created with the same name. we assume that this case is rare, and either way, identical name can imply similar functionality so the transfer of the “history” between the files isn’t completely gratuitous.   
  
**Incrementing edge-weight by action-weight:**  
Previously always increment by 1, resulting in just negating the decay (in the case where decay=1). Right now, all action weights are 1, so the change is meaningless, but we put it in place in case the action weight differ.  
  
**Splitting user’s and object’s decay**  
Meaningless in our model (since both remain with default value =1), but could be useful in the future, if optimization proves that decay for edges between objects (object\_decay) should be calculated differently from decay of edges between a user and an object (user\_decay)  
  
**Ranking all objects**  
The original model is trying to highlight the most important (according to a dynamic definition) *changes* to a given user. In our version, we’re trying to gage what the user is likely to change next. we assume that whether a file was changed by others isn’t conclusive on to itself to the question of whether a user will change it, so we rank all the objects in the graph (according to the DOI metric) to predict the next changes.  
  
**Centrality**

It the research, it was meant to be an a-priori component. In the implementation we worked from, it was an “information flow” metric for determining the node’s importance in the graph. Because of code-compatibly issues, we defined the centrality of a node to be its degree centrality (number of neighbors out of all the nodes).  
  
  
Optimizing and evaluating using Github data  
  
First, it’s worth mentioning that optimization is crucial. Since the weighted parameters (centrality, proximity and change-extent) aren’t normalized, we needed for the ratio between the weights to work as a normalization factor, as well as a measurement for the parameter’s importance.  
  
**Evaluation data**  
We used open-sourced GitHub repositories as a bench-mark, using the model to predict the files that a given user will modify in his\her next session.

We downloaded a bunch of Github repositories (full description can be found in the project files) and filtered out all the commits where a user didn’t modify existing files. We also had to filter out the repositories where the number of files exceed a curtain threshold (arbitrarily: 2000) since they proved too time-consuming. We also filtered out all files which doesn’t end in of the following: ("c", "h", "cpp", "hpp", "py", "cs") since we assumed they’re not relevant for our purpose.

At first, we tried predicting on all the data in the following method:

1. Build a base-line graph for every repository, comprised of the first 80% of the commits.
2. For the remaining 20%, at each iteration, predict the most likely objects to change and update the graph with the real changes.

Later, we figured that looking at the full life-cycle could provide more insights and moved to evaluate the graph and model on every repository starting from the first commit.

Also, we isolated the model’s results per-user, marked a couple of users as “super-users” because they were prevalence in the data (made a lot of commits) and looked of the results for those users to test the hypothesis that the model is more suited to handle frequently re-occurring entities.

**Evaluation metric**

We looked at different ways of evaluating. At first, we tried remaining close to the type of evaluations described in the paper and maintaining a notion of “top-3”, “top-5” etc. quickly we realized that it’s not applicable to this case, first because of the data (some - if not most – of the commits modified only one or two files making it kind of a binary prediction) and also because of the lack of the internal order between the modified files.

Instead, we decided to try and maximize the proportional sum of the DOI of the changed objects compared to the other objects in the graph. The idea is, if we can find the weights where the changed objects get a “larger” proportion of the total DOI, then individual changed object will be more likely to score high DOI compare to non-changed objects. But when we tried using such metrics, it became clear that they don’t work well when applied to aggregation of many commits (and many repositories) in the score function.

We also tried to find ways to expand the metric beyond the “next commit” harsh threshold, but couldn’t find a way to do it without compromising the consistency of the graph and predictions (since commits of the same user aren’t always done together, this could’ve caused silly situations where we combine two commits that were omitted a month apart because there wasn’t a commit of another user in the middle, while separating two other commits despite being made the same day).

Eventually, we settled on trying by-hand different permutations of weights and look for ourselves if there is a notable difference in the model’s results.

Results

In the “results” folder it’s possible to see visualizations and summaries for all the different attempts. In short, the models seem pretty “noisy” and not actually predicting anything or encapsulate any relevant information.

The lack of trusted “score” function made it difficult to properly compare results, but we looked at the “top-3” (correct prediction out of the 3 objects with the highest DOI), “top-5”, etc. and didn’t found much of a difference between the different combinations of hyper-parameters or notable changes in the performance for specific users.

By visualizing the MIP-graph in specific points, we tried inferring other conclusions from the graph “shape” and the ways the graph changes over time. We couldn’t discover any new information. Other than the obvious (at the beginning, the graph grows until all the files are committed and from this point on the changes are just in the connections\weights. Dah), the graphs looked to us completely descriptive. The changes were according to the pre-determined formulas, and nothing in the structure looked like it contains any extra information.

Why it didn’t work

**Not a good proxy**

One of our early conclusions from the results is that this proxy undermines the premise of the original MIP-Net model, since “interest” doesn’t equal “desire to modify”. For example, let’s assume that our repository consists of a main file and many others utility files (a structure that resembles this of most real-life repositories). And let’s assume that each team-member (unique user) is working on one of the sub-modules (again, very common in practice), a change in the main file is very relevant to every user, but our model isn’t built to recognize that fact.

This alone isn’t detrimental onto itself. It just means that we are trying to apply the MIP-Net model for a different purpose then it’s original goal. instead of measuring “importance of changes” we’re trying to predict “what a person will change”. It’s not clear if the same model will work for this different task, and we couldn’t even come up with a hypothesis because of the second problem:

**Bad idea for a data set**

Looking into the data and particularly examining the “commits-chain” gave pause to our original assumption that Github repositories can model the real-world setting of group projects.

Problems we hypothesize arise from the Github data:

1. The biggest fault of all: we can’t take the linear-order of the commits-chain as satisfying the assumption of a quasi-collaborative work environment. May times commits are made on separate branches, so the user is separated from the main work-flow, until his\her actions are retroactively omitted to the main branch. Also, commit isn’t actual pull\push, so we can’t assume that the changes were made to the repository in the order they are reported and can’t answer the basic question for this model: “who knew what and when?”.
2. The life-cycle of a repository is hardly one-size-fits-all. Some repositories are the product of gradual work (thus, emulating the settings we’re aiming for), while others are already “stabilized”. Meaning, the product\library is done and so is most of the “heavy lifting”. From that moment on, changes are made based on bugs and requests and don’t necessarily represent “interest”.
3. There is one or more person that control the commits omitted into the main branch. What ended up as part of the commits chain hardly reflect the total body of work of a given user, thus, contain only partial information regarding his\hers interests.
4. Open-source projects are notoriously problematic. Long tail, multiple consecutive changes to the same file(s), very large “background” work that don’t make it to main branch, and more.
5. A lot more problems, particularly in work-style, that aren’t accounted for. (in this project, for instance, Uriel is a known to make many commits, sometimes containing only minor changes, while Mendi is stingier, committing many changes at once every couple of days). Without fully recognizing those affects, we can’t properly negate them in our model.

We tried isolating some of those differences and looked at specific cases. For example, we looked at repositories with “long tails” in comparison to the most dominant users in a given repository. We also examined “small repositories” compared to “big”.

To us it seems that there are too many features that need to be accounted for in order to make the Github data fit the model’s assumptions, and we couldn’t even figure out how some of those features are manifested in the results.

From all of these and other theories, we strongly believe that using Github repositories as data points require deeper understanding of their structure and policies, and that different “phases” and differences between user’s role and actions are large enough to not be “smoothed” even in a large enough sample.

In fact, that realization made any attempt to better our model feels futile, since we strongly believe that measuring it using the “next file to change” proxy with the Github data without extensive modifications will result in just noise.

**Important point of emphasis**

Despite the conclusions depicted here, we do believe that the MIP-Net approach can be successfully applied to the environment of collaborative work on a programing project. It’s just that the way we tried implementing and testing it, particularly using Github repositories as a reference point, is not suitable with this model.

We believe that examining the model in a real-life simulation (in a study like the one described in the research paper) could provide some useful insights and pave the way for successful implementation.

Further exploration  
 **Problem solving**  
As mentioned above, further advancement is dependent on getting a reliable data for measuring performance. But even with a close study with a controlled group, some problems still stand. Mainly, defining what is an “important” change to flag, how to measure success or failure, how to optimize the weights and more.  
 **Trying it with namespaces (functions)**The objects in our project are defined to be documents (files) in each project. The same principles can be applied in a more granular way, by looking at each function\class as an object.

We didn’t really put much of an effort in this route, after early tries made it look like quite a taxing task (from the perspective of parsing and structuring the Github data), and we didn’t see much of an added value since we didn’t used the content of the objects anyway.

The *CsrCode* class in the CSR.py file can be used as a base class for this exercise. All other modules should generalize without (too many) changes.  
  
**Using the code-structure for extra information**So far, we treated the files as “documents” without regarding their content as significant. The structure of a programing language can provide valuable information if parsed and evaluated correctly. Most notably, it’s possible to look at the AST and defined structural relationships between the objects. Those relationships could be incorporated as extra weights in the MIP-Net or exist in parallel and taken as an extra parameter when evaluating “closeness” between nodes. This is just an example. The code structure could also tell us how many other files call or being called from the changed file (other measurement for centrality), etc.  
  
**Diving into the code**Analyzing the code-changes could uncover more information. Particularly, devising a policy to determine each action weight according to the type and size of the changes in the commit. Examples of parameters for such policy: the number of lines in the “diff” string of the changed lines, the file where the changes were made, the “type” of the change (renaming a variable vs added input parameters, etc.).  
  
**Centrality could contain actual a-priory information**We think that degree-based centrality should have a pretty week significance to determine DOI, since in a big enough project, the graph is so large that the values of this component are insignificant. Instead, it’s possible to assign the nodes real a-priori information, either from the code-structure as discussed before or from other known information and features.  
  
**The time-stamp could tell us a lot**One of the problems we mentioned above was the non-linearity of the Github data. It’s possible to remove some of the haze by looking at the time-stamp of the commit and changing the model’s behavior accordingly. One of the things we tried was adding a “time-based decay” to the model, but that’s not enough. It’s easy to think of many ways where the time passed between iterations is as (or even more) important than other parameters. **Change Extent**:As currently defined, only considers the frequency of change, not the amount of it. Maybe this metric should also incorporate the weights of the changes made to the object since the last visit of the user. **More granular decay**As mentioned above, the decay was separated for user-object and object-object decay. Right now, both have the default value of 1, since it was meaningless to optimize them without settling on the model’s weights. In the future, those parameters could be de diversified, both by changing their initial values, incorporating other parameters (like the time, as discussed), or making them dynamic (it’s stand to reason that the extent of the decay should depend on the extent of the changes that were made, and not just the number of sessions).  
  
  
What else  
  
**Machine-learning approach**  
Although we like to think of this model as pretty “smart”, the described task looks perfect for trying a machine-learning approach. Our work on the infrastructure for the retrieval and cleaning of the data, combined with some hand-crafted features, some of which already embedded in the model, make it very easy to try and predict the next code-changes using classic machine learning algorithms. Results and conclusions from these experiments could also be used to update this version of the model. For example: the weights calculated by a logistic regression could help determine the weights for the parameters in the Mip-Net model, etc.  
  
**Re-purposing the model**Our model is fairly general and can easily be applied to Wikipedia edits (for example). It’ll be interesting to see how it does in other domains. This kind of exercise could also shed light on the poor performance on the Github task and help uncover if there is some problem with the model itself, the evaluation method, or is it just bad fit for this domain.  
  
**One more recommendation**Although the original paper described a model for improving collaborative team-work, our mission seems more like a typical recommendations problem. We used the collaborative nature as a tool for discovering connections, but eventually used those connections and predicted “what this person will find the most interesting”. This is a well-studied problem and possible solutions can be found in all areas of AI. We think it could be beneficial to infuse between the two problem. Meaning, using recommendations algorithms to predict code changes (as mention above), and using the MIP-Net approach on classic recommendations datasets (the most obvious application that come to mind is something like the Facebook newsfeed, where stories are objects and profiles are the users).What we’ve learnedWell, as a research project, this can be described as a complete failure. Not only that we didn’t solve the problem we set out to solve, but the model we created looks to be pretty much useless.As a learning exercise, we took in a lot. Diving into the research paper gave us new perspective on problem solving in the AI domain (not everything is deep learning, apparently). Using graphs to model interactions in a dynamic environment and quantifying and measuring something somewhat “amorphic” as “Degree of interest” are examples for things we didn’t encounter until this project.From the discouraging results we took a lesson to be more prepared and work completely different in future projects. We rushed to the code and spent too much time on the “production” problems such as generating data, optimizing resources and so on, that we barely took the time to think of the methodology itself. Looking back, we could have predicted at least some the problems with using Github repositories. We should’ve spent more time on properly defining what we are trying to achieve and assessing whether this model combined with the available resources is best suited for this goal.