Master M2 MVA 2015/2016 - Graphs in ML

TP 3 : *Large Scale Graph Learning*

By: Lilian Besson (`lilian.besson at ens-cachan.fr`).

Attached programs:

The programs for this TP are included in the `zip` archive I sent, and are regular **Python 2** programs[1].
Note: I only used the open-source part of GraphLab, sframe (which is freely available from pip).

Note:

This last TP report is quite long, because I tried to answer to every question with as much details as possible (while not being too verbose), and a few references are included. I also included two snippets of code (Figures1 2), and in appendix A a 2-page long example of output for `prop_label.py` is entirely included.

---

1 Large Scale Semi Supervised Learning

Question 1.1 : *Give a very high level description of Data Graph, Update function and Sync function*

Shortly, we can explain the main three components of the parallel and distributed paradigm implemented in GraphLab:

- **Data Graph**: represents a sparse graph, which can be separated conceptually between user modifiable program state (e.g. `graph.vertices['f_old']` can be directly modified in `label_prop.py`) and the sparse data-structure (which aims to both store the mutable user-defined data and encode the sparse computational dependencies).

  "Data" refers to values associated with each node (e.g. out-degree or weighted degree of a node i) or each edge (e.g. distance or similarity or weight of an edge $i \sim j$ or $i \mapsto j$). In the GraphLab paradigm, the graph structure itself is static (un-mutable) while the data graph can be modified. For example, in `label_prop.py`, we see that when we added edges to make the graph undirected, we created a fresh new graph and did not modified `graph`.

- A **Update function** is a bunch of *local* computation on the graph (e.g. computing a local average[2] of a certain quantity of a node's neighbors). It represents some *factorized* (as it only needs to have a local access) user computation and operate on the data graph by transforming data in each local window *in parallel* (or at least, that's the goal).

- On the other hand, a **Sync function** is used to be able to also make *global* computation on the graph (e.g. computing the highest out-degree?), and as the documentation says it is used to maintain "global aggregate statistics" of the data graph. I think we did not used any sync function in `label_prop.py`.

> More references: More details can be found on this paper:
>
> - "The GraphLab abstraction", a paper from the GraphLab team (Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, and Carlos Guestrin, at CMU).
> - Or on the official GraphLab documentation: SGraph's doc (`graphlab.SGraph.html`), or SFrame's doc (`graphlab.SFrame.html`).

[1] They were written and tested with Python `v2.7+`, on Ubuntu (Linux) only, but everything should work on other platforms, and on Python 3 too (I'm careful with both `print` and division `/`).

[2] Exactly what is done for label propagation below (Figure 1), with `graph.triple_apply(propagate_label, "f_new")`.

2 Large Scale Label Propagation

Remark: ***Are our database large enough?*** The TP was advertising the use of GraphLab for handling *large* database ("GB-size databases"), but here the movies database weight 168 KB and just 3900 lines, and the ratings database is bigger with 24 MB and about 1 million lines. But none of them seemed too huge to fit in the RAM...

(However, this TP was just a practical session, its goal was not to implement a real-world recommendation system and so it was clever to only work with "middle-scale" database).

In fact, I was surprised[3] to see that the GraphLab server and the SFrame were using **a lot** of RAM, way more than the size of the raw database files, and that's logical: storing a raw file takes less space than storing it in a "smart" way to be able to read from and write to it!

Question 2.1 (a) : *What is the main drawback of storing data on the disk?*

Storing a huge graph on the disk and not on the RAM of our machine is a good idea to try to scale to bigger databases (GB or TB sized).

However, initially the RAM was introduce to reduce what is called the IO wait: the CPU (computation time) is way quicker than the input/output with the disk. Using the RAM allows to reduce this "IO wait", the time when the CPU has to wait for some data to be read from or written to the memory.

Therefore, storing our data on the disk brings back this old problem. The CPU might spend 99.9% of is computation time waiting for the new data to be fully written to the disk (or fully read)...

Question 2.1 (b) : *How can we try to mitigate this drawback when implementing our algorithms?*

The first idea we can try is to reduce as much as possible the number of access (read or write) to the data. We can try to access or to read bunch of data at the same time.

We can also try to be "clever" in the way the data is stored, and try to respect some specific geometry of the data (in the case of a graph, store a node's neighbors' data "close" to the node's data, close in the sense of quicker IO access). For example, this is what is done by GPGPU (graphical cards) used for Computation Fluid Dynamics: the matrix has a 2D or 3D geometry, which is used to cleverly parallelize the computation.

Question 2.2 : *Can you think of a simple problem with using mutable variables in a closure in a distributed framework like GraphLab?*

A distributed framework (like GraphLab) will, as his name suggests it, perform computations in a *distributed* and *parallel* way.

With rough notations, the parallel aspect simply say that the same computation c (for instance, computing $f(j)w_{i,j}/\sum_j w_{i,j}$) will be done exactly at the same time t for different vertex (or different edge, or different whatever), at least for two different entities i and k. If this computation c at time t tries to modify a mutable variable v, the modification will be done at the same time t for both i and k (and possibly many more): $v \leftarrow c_i(t)$ *and* $v \leftarrow c_k(t)$. There is no way of knowing which one will succeed (and even if any of the $c_i(t)$ will work), and the value v after these computation might be purely random, so we cannot rely on it afterward and therefore the computation were useless (or dangerous)...

> $\Longrightarrow$ *Long story short*: for these reasons, modifying in parallel a same mutable variable (or a more complex data-structure) is a *bad idea* in general.

Note: The same issue arises when using simple multi-threading (a framework for parallel computation, but simpler than GraphLab). See the Wikipedia page on Concurrent Data Structure for more details.

[3] Below (2) is explained a trick I added to the program, to reduce RAM consumption, by manually deleting variables as soon as we were done working with them – that's manual garbage collection!

Question 2.3 : *Complete* `label_prop.py`

See the program `label_prop.py`, included in the submitted `zip` file. For an example of its output, I included a file `label_prop.log`, which shows the complete output (with detailed messages) for a final relative accuracy of 11% (see Appendix A also).

For two attribute names, I felt that the notations were not perfectly clear, so they are explained below (I have not changed any naming convention):

- `degree` for a node i (`src` or `dst`) refers to the **weighted degree** $\sum_{j,i\mapsto j} w_{i,j}$ which is a weighted sum, not the usual degree which just count the number of neighbors ($d_i = \sum_{j,i\mapsto j} 1 = \#\{j/i \mapsto j\} = \#\mathcal{N}(i)$).
- `distance` for an edge $e = (i \mapsto j)$ is not really a distance but is rather an **"anti-similarity"**[4]: 0 means equal, 1 means very different. When I printed the graph's edges, I indeed observed that the `distance` field has values in $(0, 1]$, with lots of values close to 1 (ie. lots of movies are seen as quite different, and that's satisfying).

We list below some important modifications we had to do on the given skeleton:

- The two `.dat` database files are read from a **local path** (`read_csv('td3-data/movies.dat')`) and not `/home/student/td3-data/` (I did not use the VM for this TP).
- (No other important modification were done.)

Below is included the key part of that file, the `propagate_label` function (Figure 1). It is important to notice that *nothing* had to be done on `dst['f_new']` on this function, because the fact that the graph is undirected has already been taken care of when building the graph.

```
def propagate_label(src, edge, dst):
  if src['is_labeled']:
    src['f_new'] = src['f_old']
  else:
    src['f_new'] += (dst['f_old'] * edge['distance']) / src['degree']
    # If gamma regularization:
    # src['f_new'] += (dst['f_old'] * edge['distance']) / (src['degree'] +↩
        gamma)
  return (src, edge, dst)
```

Figure 1: Snippet of code for the `propagate_label` Update function.

Then we list below the "smart" things we had to do:

- I computed a `genre_dict` dictionary, and printed a short list of how many movies are labeled with each genre. We learn for example that *Drama* is the most present genre (1603/3883), and *Film Noir* is rare (only 44/3883).
- In order to reduce the (huge) RAM consumption[5] of the program, I added a few cleverly placed `del(...)`, to delete some variables as soon as we are done using them (the two SFrame, the matrix `feat_mat` etc).
- In the whole program, I printed some information messages, in order to see what was going on. I added a timer for the reading and building[6] of the two SFrame `sframe_movies` and `sframe_ratings`.

[4] As we can see, it is given when initializing the ANN engine, as a cosine similarity: `distance = nearpy.distances.CosineDistance()`.

[5] It drastically improved the "stability" of the execution (my laptop is an old one, 4Gb RAM, and it can freeze if the RAM use is too close to 100%).

[6] See below (Par. 2) for an explanation about why extracting this ratings database is probably the bottleneck of our program.

- At the end, I added a few messages to display the raw and relative accuracy. This helps to see that we got around 9% and then around 20.5% of relative accuracy after tweaking the parameters (see below in Question 2.9 2). The TP assignment gave absolutely *no idea* of what a good (relative) accuracy should be! Therefore I decided that 20.5% was good enough. I asked the question on Piazza, and aiming at 30% accuracy seems impossible.
- Finally, for each line that has been modified from the original file, a comment "`# XXX`" was added (it can help to easily see what I modified).

Where is the bottleneck of our program?

Running this program was quite long, and in fact I found that the *bottleneck* was the extraction of the `ratings` database. The slower part is not reading the `.dat` file from the disk, but building it into a SFrame. It took about 5 minutes in my machine, while the rest of the code was really quicker. In comparison, each run of `propagate_label` (using a `triple_apply`) took about 20 seconds.

Question 2.4 : *Why do we resort to Approximate Nearest Neighbors (ANN) to compute the similarity graph?*

Shortly, computing the k exact Nearest Neighbors (ENN) of a node i requires to **read the entire graph** and not some small sub-graph (as ANN does). Therefore, ENN seems to be **very hard to parallelize**.

Question 2.5 : *The implementation provided by NearPy uses Local-Sensitivity Hashing. Try to give an high-level explanation of how LSH works, and why we chose to use it in this problem compared to Tree-based ANN.*

Shortly, *hashing* is a way of projecting high-dimensional or high-cardinality data ($\in \mathcal{S}$, like $\mathbb{R}^d$ or $[1, \dots, K]$ with huge K) to a small-dimension space, ideally $\mathbb{R}^1 = \mathbb{R}$ or a finite space $[1, \dots, B]$ (with $B \ll K$). Usually, for cryptographic application, we aim at minimizing the risk of collision (hash functions are usually random, so it boils down to minimizing the probability of getting the same hash for different messages). On the other hand, Local-Sensitivity Hashing is a clever way to do this reduction of dimension, but it tries to respect locality: if $\|x - y\|_{\mathcal{S}}$ is small, $\|h(x) - h(y)\|$ should also be small! Closer input points are hashed to the same value or to a close hash value.

LSH[7] hashes the input data so that similar points map to the same "buckets" with a high probability (the number of buckets B being much smaller than the number K of possible input items or dimension d of input space). Note that it differs from conventional and cryptographic hash functions because HFS aims to *maximize* the probability of a *collision* for similar input points.

- I think the first reason why we choose LSH for our application is simply... because the ANN engine we chose to use (NearPy) only supports variants of LSH but does not support tree-based[8] ANN (yet).
- The second reason might be than LSH appears to work better than tree-based approaches for low-dimensional data but huge database (here movies or users have a very small number of components).

Question 2.6 : *Why do we choose not to use a closure when feeding the ANN Engine?*

This question is similar to Question 2.2. In fact, the issue is that we don't really know how the key function of NearPy's ANN engine (`ann_engine.store_vector`) works (and there is no documentation[9]...).

I think we choose to not use a closure in order to remove (or reduce) the risk explained in Question 2.2, ie. the risk of concurrently modifying the same mutable data-structure (a node of the ANN engine). As we don't

[7] This page details the use of LSH for approximate Nearest Neighbors search, but this goes beyond the reach of this practical session. And this paragraph lists different approaches to ANN.

[8] See this paper by David M. Mount and Sunil Arya, and their library implements several algorithms for ANN.

[9] Note that NearPy's developer is already aware of that issue, but has still not started to work on it. I would love to help him but do not have the time right now – maybe in April?

know how exactly the ANN engine is fed, it's simply safer to feed it the vectors in a sequential way (hence the `for` loop and not a `triple_apply` or a closure).

Question 2.7 : *Why do we need line* 156 *(in the original file) in* `label_prop.py`?

This question refers to this following snippet of code (Figure 2):

```
# At least 5 movies of each genre in the sub sample
nb_min_movies = 5
for i in xrange(num_movie_classes):
    l_idx += random.sample(classes_to_movie_map[i], nb_min_movies)
    # l_idx can have doublons now!

print("We added", nb_min_movies, "movies of each genre in the random ↩
    sample (to be sure)...")
print("Now there is", len(set(l_idx)), "different movies in the random ↩
    sample.")
```

Figure 2: Snippet of code to ensure that each genre is present in the random sample (with at least `nb_min_movies = 5` representatives).

This step ensure that we have at least `nb_min_movies` movies of each genre in the random sub sample of labeled movies. Indeed, if we only take a random sample of indexes (the list `l_idx`), we have a risk of getting no labeled examples for some genre. And that's why the *list* `l_idx` is then converted to a *set*: this step could have added doublons, they need to be removed.

And from the lecture, we know that any form of SSL will perform badly when trying to predict ratings for unknown movies of a genre for which no labeled examples are available! Therefore, this part is only here to be sure this won't happen.

Question 2.8 : *In normal HFS, regularization is added to the Laplacian to simulate absorption at each step in the label propagation. How can you have a similar regularization in the iterative label propagation?*

I think we can chose a very small value for γ (like 10^{-2}), and instead of dividing by $\delta_{\text{weighted}}(i)$ (= `src['degree']`) in the iterative label propagation, we divide by $\gamma + \delta_{\text{weighted}}(i)$. See the snippet of code in Figure 1. I tried this value of $\gamma = 10^{-2}$, but in practice I didn't observed a better accuracy.

Note: I am pretty sure that this was not the good way to add this $\gamma-$regularization, but I didn't find any other idea.

Question 2.9 : *Try different combinations of parameters for the neighbors, regularization, and hashing space and plot the accuracy and running times.*

My program `label_prop.py` was taking about 11 minutes each time I ran it, and so exploring a lot of different parameters (enough to make a nice plot) was just too time consuming, sorry.

Below is listed my intuition (and some observed behaviors) on the influence of each parameter:

- Increasing the number of neighbors increases the computation time and increases the accuracy (the more neighbors we can trust the better but the slower). In practice, I observed this, by trying `num_nn` $= 30, 50, 70, 80, 90, 100$. 100 gave the best result.
- Increasing the number of label movies to use (the parameter `num_sub_movies`, given = 250) clearly increase accuracy while not reducing the running times (`propagate_label` is quicker for labeled movies). But this is a normal observation: for SSL, the more labels we have, the better we will be!

- For this program, "regularization" can refer to several parameters.
  - It might refers to `min_score` (the minimum score to be considered, was given as 4, ie. we keep a film's rating only if the user rated it 4 or 5 stars out of 5). In practice, I tried using 3 or 2 but did not really see any change.
  - It might also refers to `nb_min_movies` (given = 5). In practice, I tried using $3, 5, 8, 10, 15, 20, 25$ and increasing it has almost the same consequence as increasing the number of labeled movies `num_sub_movies`. 25 gave the best result.
  - In fact, it should be understood as the γ regularization parameter. See above question 2.8 2 for an explanation of its influence.
- And of course, increasing the number of label propagation iteration (the parameter `nb_iter`, given = 3, but finally chosen to be = 25) increase the accuracy while increasing the computation time (as usual).

References: On the GraphLab documentation, I found that they already offer an implementation of label propagation, but I ran out of time to try to use it and compare it with ours. More details:

- On the documentation: GraphLab's graph analytics toolkit (`graphlab.toolkits.graph_analytics`) and its Label Propagation Model (`graphlab.label_propagation.LabelPropagationModel`) along with the appropriate function `graphlab.label_propagation.create`
- And a reference paper: "Learning from labeled and unlabeled data with label propagation", by X. Zhu, and Z. Ghahramani (2002).

---

3 Large Scale and Sparsification

Question 3.1 : *Complete* `sparsification.py`

See the program `sparsification.py`, included in the submitted `zip` file. For an example of its output, I included a file `sparsification.log`, which shows the complete output (with detailed messages) for a final average ration of 0.9087 for $\varepsilon = 0.2$ (see Appendix B also).

We chose the following values for the parameters of this random experiment:

- $n = 1000$ the size of the (random) graph used for testing the sparsification. It should be small enough for the experiment to run quickly, and big enough to "deserve"[10] to be sparsified.
- `density` $= d = 0.7$ is just the density of the initial (random) graph G. With this value, the graph G will have about 70% of the total number n^2 of possible edges (so G is already "a little bit" sparse).
- $\varepsilon = 0.2$ is the ε in Algorithm 1, it is the aimed density for the sparse approximation H of G (or L_H of L_G). With this value 0.2, we hope to save up to 80% of storage space (resp. computation time) when storing (resp. computing with) L_H instead of L_G. It is the linked with the confidence of the (approximate) sparsification.
- `num_test` $= 200$ is the number of experiment we will do. Each experiment is taking some random (Gaussian) vectors r_k ($1 \leqslant k \leqslant$ `nb_random_vector`), and compute the averaged ratio of $\frac{r_k^T L_H^+ r_k}{r_k^T L_G^+ r_k}$. We hope that this averaged ratio will between $1-\varepsilon$ and $1+\varepsilon$ (cf. equation (2) in the assignment, page 9).
- `nb_random_vector` $= 100$ is the number of random vectors r we sample for each experiment. We choose is to be 100 because what matters is the total number of random vectors we tried, which obviously has to be big (`num_test` $\times$ `nb_random_vector` $= 20000$).

[10] Sparsification for small matrices or graphs is known to be almost useless, or worse: usually smart sparse data-structures use more memory than naive data-structures for small data.

See below (Algorithm 1) for the Spielman-Srivastava Sparsification algorithm[11] implemented in the `select_number_of_copies` function:

Data: $G = (V, E)$, N, ε, and effective resistances $R_{e e \in E}$
Result: H
Set H to be the empty graph on V (ie. $H = (V, \varnothing)$);
for $i = 1$ *to* $N = O(n \log(n/\varepsilon^2))$ **do**
  Choose a random edge $e \in E$;
  Set $p_e \propto R_e$; `/* (with a known constant, e.g. 1/(n-1)) */`
  if *With probability* p_e **then**
    Add e to H with weight $w_e = 1/(N \times p_e)$
  end
end

Algorithm 1: Spielman-Srivastava Sparsification algorithm.

As suggested by the assignment, what is finally displayed by the program is the average ratio between the sparsifier L_H and the original graph (L_G) on a set of random vectors. For the parameters chosen as above, we got `avg_ratio` $= 0.908$. This is a very satisfying result, as the goal of this average ratio was to be between $1-\varepsilon$ and $1+\varepsilon$, and with $\varepsilon = 0.2$ we have what we wanted. I think with $n = 1000$ and these choices of parameters, we tried a setting "big enough" to be happy of this final result.

Question 3.2 : *How can you exploit the structure of b_e to extract R_e efficiently from L_G^+ ?*

Let $e = (i \sim j)$ an edge of the graph G, $\overrightarrow{E_i}$ indicator vector for the node i, and $b_e = \overrightarrow{E_i} - \overrightarrow{E_j}$.

If $R_e = b_e^T . L_G^+ . b_e$, we just expand b_e to get:

$$\begin{aligned} R_e &= b_e^T . L_G^+ . b_e \\ R_e &= \left(\overrightarrow{E_i} - \overrightarrow{E_j}\right)^T . L_G^+ . \left(\overrightarrow{E_i} - \overrightarrow{E_j}\right) \\ R_e &= (\overrightarrow{E_i}^T . L_G^+ . \overrightarrow{E_i}) - (\overrightarrow{E_i}^T . L_G^+ . \overrightarrow{E_j}) - (\overrightarrow{E_j}^T . L_G^+ . \overrightarrow{E_i}) + (\overrightarrow{E_j}^T . L_G^+ . \overrightarrow{E_j}) \end{aligned}$$

But these dot product can be simplified: $\overrightarrow{E_i}^T . A . \overrightarrow{E_j} = A_{(i,j)}$. And if A is symmetric $A_{(i,j)} = A_{(j,i)}$:

$$R_e = \left(L_G^+\right)_{(i,i)} + \left(L_G^+\right)_{(j,j)} - 2\left(L_G^+\right)_{(i,j)}$$

So we have a very simple formula for the effective resistance R_e of an edge $e = (i \sim j)$:

$$R_e = \left(L_G^+\right)_{(i,i)} + \left(L_G^+\right)_{(j,j)} - 2\left(L_G^+\right)_{(i,j)}.$$

It is symmetric in i, j, as expected, and is 0 for $i = j$ (no resistance on a self-loop, this is logical too). And this formula takes a time $O(1)$ to compute, way quicker than a double dot product (in $O(n)$).

Warning: this formula uses L^+, not directly L (L^+ is its pseudo-inverse[12]). As explained in Question 3.4 (3), this matrix (pseudo-)inversion is computationally heavy!

Question 3.3 : *How can you implement the inner loop implicitly?*

Instead of manually repeating N Bernoulli trials (ie. $Bernoulli(p_e)$), we can directly use a Binomial (ie. $Bin(N, p_e)$).

[11] The reference paper is "Graph Sparsification by Effective Resistances", by Daniel A. Spielman and Nikhil Srivastava, 2009.
[12] See Section 2.2 (page 5) of "Graph Sparsification by Effective Resistances", *The Pseudoinverse*, for more details.

In fact, it's what I did afterwards, to compare the running time between our manual N Bernoulli trials and `numpy.random`'s or `scipy.stats` implementation of the Binomial distribution. In practice, as expected, using the `numpy.random.binomial` function seems quicker.

Question 3.4 : *How to use an efficient approximated Least-Square solver to speed up extraction of effective resistances?*

Instead of inverting G's Laplacian or computing its pseudo-inverse (matrix $L_G \mapsto$ matrix L_G^+, as described above in Question 3.2 3), to solve the underlying linear system which gives the effective resistance R_e, we could try to use this Least-Square solver to approximatively solve the linear system, and therefore drastically speed up the extraction of R_e.

This is far from being trivial, and it is explained in detailed in section 4 (page 10) of the "Graph Sparsification by Effective Resistances" paper, *Computing Approximate Resistances Quickly*. The key idea is to compute an approximation of $R_e = \|W^{1/2}BL^+(b_e)^2\|_2^2$, and this is done with the efficient LS solver STSolve[13]. In fact, they do something strong, by building an intermediate data-structure (a matrix $\tilde{Z}$ of size $nO(\log n)$) from which the effective resistance R_e can be efficiently approximated (in time $O(\log n)$).

Were we too optimistic?

In fact, we cannot have a generic LS solver running in $O(m\log(n))$. I checked quickly, from the Convex Optimization course given by Alexandre d'Aspremont, and this complexity seems too optimistic. But for special cases, for Laplacians (or strongly diagonally dominant systems) we can indeed hope for a more efficient solver[14]. But anyway, the question was starting with "If I could provide you", so we don't really care if it is realistic or not to expect such an efficient solver.

---

4 Conclusion

Remark: Again, we found this last TP to be quite long...
But this introduction to smart parallel data-structure like SFrame and SGraph was interesting, thanks!

---

[13] Reference papers are "Spectral Sparsification of Graphs" and "Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems", by D.A. Spielman and S.-H. Teng, 2006 and 2008.

[14] A reference for a LS solver running in just $O(m\log(n))$ is this paper "A nearly-$m\log(n)$ time solver for SDD linear systems" by I. Koutis, G. Miller and R. Peng.

A Appendix : embedding of a `label_prop.log` file

Below is included an example of output for the `label_prop.py` program, showing the different steps on the execution, and its final score is 20.06% (it is the best one I got).

```
Starting label_prop.py ...
We chose a gamma = 0 (regularization parameter)
Loading the movies database...
PROGRESS: Finished parsing file ./td3-data/movies.dat
PROGRESS: Parsing completed. Parsed 3883 lines in 0.106086 secs.
Done loading the movies database, it took 0.134 seconds ...
There is 18 genres of movies.
For each genre, print number of movies :
  - There is 106 movies of the genre Mystery
  - There is 276 movies of the genre Sci-Fi
  - There is 211 movies of the genre Crime
  - There is 1603 movies of the genre Drama
  - There is 251 movies of the genre Children's
  - There is 105 movies of the genre Animation
  - There is 503 movies of the genre Action
  - There is 1200 movies of the genre Comedy
  - There is 127 movies of the genre Documentary
  - There is 114 movies of the genre Musical
  - There is 471 movies of the genre Romance
  - There is 343 movies of the genre Horror
  - There is 44 movies of the genre Film-Noir
  - There is 143 movies of the genre War
  - There is 68 movies of the genre Fantasy
  - There is 283 movies of the genre Adventure
  - There is 492 movies of the genre Thriller
  - There is 68 movies of the genre Western
  - There is 3883 different movies.
We take a random sample of size 250
  First 10 values of the random sample: [2621, 3659, 218, 3377, 2819, 2595, 1544, 1219, 16, 34]
We added 25 movies of each genre in the random sample (to be sure)...
Now there is 635 movies in the random sample.
Loading the ratings database...
PROGRESS: Finished parsing file ./td3-data/ratings.dat
PROGRESS: Parsing completed. Parsed 1000209 lines in 1.74542 secs.
Done loading the ratings database, it took 2.065 seconds ...
Looking at the movies with a score >=  4 stars.
There is 6040 different users in the database.
We had 3883 movies, but only 3706 are rated.
    (So 4.56% are not rated...)
Using 100 nearest neighbours.
Initializing the ANN engine (from nearpy)...
Extracting features from the SFrame sf_features...
Feeding the ANN engine...
Creating edge list...
We have created 33458 edges...
Building the graph, first with 33458 edges...
We make it undirected, so it now has twice as much edges, indeed it has 66916 edges...
Adding an attribute 'is_labeled'...
Initializing old and next labels (f_old <- movie_desc, f_new <- fresh zeros matrix)...
Starting iterations to propagate labels... nb_iter = 25
       Iteration number 1 ...
```

```
       Propagating labels...
       Updating labels (f_old <- f_new)...
       Reinitializing next labels (f_new <- fresh zeros matrix)...
    Done iteration number 1, it took 37.38 seconds ...
       Iteration number 2 ...
       Propagating labels...
       Updating labels (f_old <- f_new)...
       Reinitializing next labels (f_new <- fresh zeros matrix)...
    Done iteration number 2, it took 32.08 seconds ...
...
       Iteration number 25 ...
       Propagating labels...
       Updating labels (f_old <- f_new)...
       Reinitializing next labels (f_new <- fresh zeros matrix)...
    Done iteration number 25, it took 21.76 seconds ...
Done for the labels propagation... nb_iter = 25
Evaluating the final labeling...
Reading ground truth from the database.
Evaluating performance...
Done, raw accuracy = 3104
So with 3883 movies, the percentage accuracy = 20.06181%.
So far, the best I got was 18.5%...
Done for label_prop.py !
```

B Appendix : embedding of a `sparsification.log` file

Below is included an example of output for the `sparsification.py` program, showing the different steps on the execution, and its final average ratio is 0.91, which is coherent : we were excepting something of the order of $1 \pm \varepsilon$ and $\varepsilon = 0.20$.

```
Starting sparsification.py ...
Metaparameter for the sparsification experiment :
    - epsilon = 0.2,
    - density = 0.7,
    - n = 1000,
    - num_test = 200,
    - nb_random_vector = 100
Generate the random matrix G, of size 1000x1000, and density d = 0.7 (so of size 700000.0)...
G has been created, and made symmetric...
Building its Laplacian, L_G...
Building the inverse of his Laplacian, Li...
Computing the inverse of L (variable Li)
Creating a GraphLab's SGraph graph...
Creating an empty edge list...
Computing the effective resistances... and adding edges
Done computing the effective resistances...
For example, the last one was 0.00279235839844.
Adding all these edges to the GraphLab's SGraph graph...
Initializing weights to 0...
Starting to build the sparse graph... (with graph.triple_apply)...
Creating a list of sparse edges...
We have (253657, 3) edges in this list.
Converting to a matrix, and creating H.
Matrix of sparse edges, of size (253657, 3).
Adding sparse edges to matrix H (of size 1000x1000)
Making H symmetric!
Building its Laplacian, L_H...
Building the inverse of his Laplacian, Lh_inv...
Only computing the pseudo inverse... warning ! (variable Lh_inv)
Final edges: 254156
Starting to compute the final evaluation, with an average ratio...
Starting 200 experiments...
Done with these 200 experiments (each using 100 random vectors).
Before being normalized, the total summed ratio is: 18174
We divide it by the total number of random vectors 200x100 = 20000
Average ratio: 0.9087
```