TP3: Large Scale Graph Learning

daniele.calandriello@inria.fr

Monday 28th November, 2016

Abstract

The report and the code are due in 2 weeks (deadline 23:59 12/12/2016). You will find instructions on how to submit the report on piazza, as well as the policies for scoring and late submissions. All the code related to the TD must be submitted. *Material on* `http://chercheurs.lille.inria.fr/~calandri/`

A small preface (especially for those that will not be present at the TD). Some of the experiments that will be presented during this TD makes use of randomly generated datasets. Because of this, there is always the possibility that a single run will be not representative of the usual outcome. Randomness in the data is common in Machine Learning, and managing this randomness is important. Proper experimental setting calls for repeated experiments and confidence intervals. In this case, it will be sufficient to repeat each experiment multiple times and visually see if there is large variations (some experiments are designed exactly to show this variations).

Whenever the instructions require you to complete a file (e.g. `exponential_euclidean.m`) this means to open the corresponding file, and complete it according to the instructions all the following sections

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% use the build_similarity_graph function to build the graph W  %
% W: (n x n) dimensional matrix representing                     %
%    the adjacency matrix of the graph                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

YOUR CODE HERE

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

For each of these file the documentation at the top describes their parameters.

1 Large Scale Semi Supervised Learning

A large part of the algorithms on graph that we have seen so far, depends on matrix representation of the graph (e.g. Laplacian) to compute their results. This means that the execution of these algorithm is strongly influenced by constraint of representing matrices in memory, and the computational complexity of executing operations on them.

We will see later how the computational complexity plays a role, but the largest constraint in real-world systems is memory occupation. In particular, storing in memory all the elements of an $n \times n$ matrix will require around 190 MB for $n = 5000$, but will easily cross the GB threshold at $n = 12000$ and the 10 GB threshold at $n = 37000$. At this point, storing the whole graph in a single machine's memory becomes impossible, and we must either compress it using a graph approximation technique, distribute it across several machine, or store it in a larger memory and process it in multiple passes.

2 Online SSL

Semi-Supervised Learning was introduced as a solution designed for problems where collecting large quantities of Supervised training data (usually labels) is not possible. On the other hand, in SSL scenarios it is usually inexpensive to obtain more samples coming from the same process that generated the labeled samples, but without a label attached. A simple approach is to collect as much additional data as possible, and then run our algorithm on all of it, in a batch or off-line manner. But in other cases it is possible to start the learning process as early as possible, and then refine the solution as the quantity of available information increases. An important example of this approach is stream processing. In this setting, a few labeled examples are provided in advance and set the initial bias of the system while unlabeled examples are gathered online and update the bias continuously. In the online setting, learning is viewed as a repeated game against a potentially adversarial nature. At each step t of this game, we observe an example x_t , and then predict its label f_t . The challenge of the game is that after the game started we do not observe any more true label y_t. Thus, if we want to adapt to changes in the environment, we have to rely on indirect forms of feedback, such as the structure of data. Another difficulty posed by online learning is computational cost. In particular, when t becomes large, a naive approach to hard-HFS, such as recomputing the whole solution from scratch, has prohibitive computational costs. Especially in streaming settings where near real-time requirements are common, it is imperative to have an approach that has scalable time and memory costs. Because most operations related to hard-HFS scale with the number of nodes, one simple but effective approach to scale this algorithm is subsampling, or in other words to compute an approximate solution on a smaller subset of the data in a way that generalizes well to the whole dataset. Several techniques can give different guarantees for the approximation. As you saw in class, incremental k-centers [1] guarantees on the distortion introduced by the approximation allows us to provide theoretical guarantees.

Algorithm 1 Incremental k-centers

```
1: **Input:** an unlabeled $\mathbf{x}_t$, a set of centroids $C_{t-1}$, $\mathbf{p}^c, \mathbf{p}^n, \mathbf{b}$
2: **if** $(|C_{t-1}| = k + 1)$ **then**
3:     Find two closest centroids $c_{add}$ and $c_{rep}$. $c_{add}$ will forget the old centroid and will point to the new sample that just arrived, and $c_{rep}$ will take care of representing all nodes that belonged to $c_{add}$
4:     If necessary $R \leftarrow \sigma R$. If this happens, reset the taboos to only the labeled nodes.
5:     All $p_i^n$ that pointed to $c_{add}$ must be updated. To find the new node that they must point to, we use $\mathbf{p}^c$.
6:     We mark $c_{rep}$ as taboo, to avoid merging together too many close centroids.
7:     $c_{add}$ will now represent the new node, we must update accordingly $\mathbf{p}^c, \mathbf{p}^n$ and the stored centroids.
8: **else**
9:     $C_t \leftarrow C_{t-1}$
10:    $x_t$ is added as a new centroid $c_t$ and $\mathbf{p}^c, \mathbf{p}^n$ are updated accordingly
11: **end if**
```

There is a lot of practical consideration to keep in mind when implementing this kind of algorithms.

- The labeled nodes are fundamentally different from unlabeled ones. Because of this, it is always a good idea to keep them separate, and never merge them in a centroid. In the implementation this is accomplished with a taboo list **b** that keeps track of nodes that cannot be merged.
- Altought the doubling algorithm has this name, 2 is not always the optimal constant to increase the R constant, neither theoretically or in implementation. A large R factor provides worse performance, so we should set it as low as possible as long as we can guarantee the invariants.
- Ammortized cost matters, especially in streaming application. It is not always possible to stop execution to repartition the centroids, and it is often preferrable to pay a small price at every step to keep execution smooth. In our case, the centroids are updated at every step.

The most important variables for the centroid update are the `centroids_to_nodes_map` and the `nodes_to_centroids_map`. They allow us to extract labels from the solutions, and to correctly mantain information on the multiplicities of every centroid. `centroids_to_nodes_map` $\mathbf{p}^c$ is a k dimensional vector such that p^c_i is the index of the nodes representative of the i-th centroid. On the other hand `nodes_to_centroids_map` $\mathbf{p}^n$ is the t dimensional vector that maps every node to the centroid that represents him. Whenever a new node arrives, and we have too many centroids, we choose the two closest centroids c_{add} and c_{rep}. c_{add} will forget the old centroid and will point to the new sample that just arrived, and c_{rep} will take care of representing all nodes that belonged to c_{add}.

2.1. Complete `online_ssl_update_centroids.m` using the pseudocode from 1.

Computing the solutions from the quantized graph using hard (constrained) HFS is pretty straightforward.

Algorithm 2 Online HFS with Graph Quantization

```
1: Input: $t$, a set of centroids $C_{t-1}$, $\mathbf{p}^c, \mathbf{p}^n, \mathbf{b}$, labels $\mathbf{y}$
2: Compute $L_t$ of $G$ using $W_q = V\widetilde{W}_q V$
3: Infer labels using hard-HFS
4: Predict $\widehat{y}_t = \text{sgn}(\mathbf{f}_u(t))$.
5: With the preceding construction of the centroids, $x_t$ is always present in the
   reduced graph and does not share the centroid with any other node.
```

2.2. Complete `online_ssl_compute_solution.m` following the pseudocode from 2

Use `create_user_profile.m(stream,'name_of_subject')` to capture a training set of labeled data of your face and someone else. Read the help for `cv.VideoCapture` to understand how to open a stream. The faces will be preprocessed and saved in the same folder as `name_of_subject[1,2,3,...].bmp`. They will be loaded by `online_face_recognition.m` (you have to adjust the subjects name in the file).

2.3. Complete `preprocess_face.m` and run `online_face_recognition.m` either live or on a previously recorded video. Include some of the resulting frames (not too similar) in the report showing faces correctly being labeled as opposite, and comment on the choices you made during the implementation.

2.4. What happens if an unknown person's face is captured by the camera? Modify your code to be able to disregard faces it cannot recognize, and include some of the resulting frames (not too similar) in the report showing unknown faces correctly being labeled as unknown.

3 Large Scale Label Propagation

We will begin by implementing label propagation on a middle scale dataset. In TD2 we have seen how to do this by computing an HFS solution in closed form. The Harmonic property that the HFS solution wants to satisfy is

$$f(x_i) = \frac{\sum_{i\sim j} f(x_j) w_{ij}}{\sum_{i\sim j} w_{ij}}$$

Computing HFS using linear algebra is not particularly easy to parallelize, and does not decompose easily over vertices as required by most distributed graph frameworks. But the Harmonic property gives us an easy idea for implementing this algorithm iteratively with subsequent sweeps across the graph.

For the implementation we will use Octave/Matlab's, which represent a graph using sparse matrices in Compressed Sparse Column (CSC) format[1]. When implementing an algorithm, it is often important to have at least a rough idea of the underlying data representation used by the implementation. For CSC matrices, accessing the matrix column-wise is a much cheaper operation than extracting a row of the matrix. Another common constraint for distributed algorithms is that when the algorithm processes node i, it has access only to the edges incident to node i itself. We will represent this constraint by allowing only operations on single columns of the matrix in `iterative_hfs.m`, forbidding global operations on the whole matrix (e.g. matrix multiplication).

3.1. Complete `iterative_hfs.m`. Be careful to only use vector operations (e.g. vector-vector multiplication, sum over a single colum), and to respect the column-wise access pattern suggested by CSC.

3.2. 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?

4 Distributed Graph Computing

Another important part of computation on large scale is parallelism. Parallel algorithm have the advantage of distributing the data across nodes to increase memory capacity, as well as computing different parts of the solution concurrently on different nodes. During class, we presented the Graphlab[2] library.

[1] https://en.wikipedia.org/wiki/Sparse_matrix
[2] https://turi.com

Graphlab implements a parallel, distributed paradigm for implementing machine learning algorithms, and hides most of the complexity of the parallelization and communication from the user. Nonetheless, it is important to have an idea of how a modern parallel system abstract its computations

4.1. Briefly read Section 2 (2.1, 2.2 and 2.3) of `http://select.cs.cmu.edu/code/graphlab/abstractiononly.pdf` and give a very high level description of Data Graph, Update function and Sync function

One of the main objective of Graphlab is allowing computation to be carried out not only on datasets that can fit in memory. When a cluster of machines is not available, GraphLab stores most of the `SGraph`'s data on disk, and efficiently maps it into memory when a function has to be applied on it. This allows even normal machine to scale to multi-GB sized datasets.

4.2. What is the main drawback of storing data on the disk? How can we try to mitigate this drawback when implementing our algorithms?

References

[1] Moses CHARIKAR, Chandra CHEKURI, Tomas FEDER, and Rajeev MOTWANI. Incremental clustering and dynamic information retrieval. *SIAM journal on computing*, 33(6):1417–1440, 2004.