Graphs in Machine Learning

Michal Valko

*Inria Lille - Nord Europe, France*

TA: Daniele Calandriello

Partially based on material by: Tomáš Kocák, Nikhil Srivastava, Yiannis Koutis, Joshua Batson, Daniel Spielman
Last Lecture

- Examples of applications of online SSL
- Analysis of online SSL
- SSL Learnability
- When does graph-based SSL provably help?
- Scaling harmonic functions to millions of samples
This Lecture

- Large-scale graph construction and processing (in class)
- Scalable algorithms:
  - Graph sparsification (presented in class)
  - Online face recognizer (to code in Matlab)
  - Iterative label propagation (to code in Matlab)
This Lecture/Lab Session

▶ AR: record a video with faces
▶ Short written report
▶ Questions to piazza
▶ http://researchers.lille.inria.fr/~calandri/teaching.html
Large scale Machine Learning on Graphs

http://blog.carsten-eickhoff.com

Botstein et al.
Are we large yet?

<table>
<thead>
<tr>
<th>App</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>1,712</td>
</tr>
<tr>
<td>WhatsApp</td>
<td>1,000</td>
</tr>
<tr>
<td>Facebook Messenger</td>
<td>1,000</td>
</tr>
<tr>
<td>QQ</td>
<td>899</td>
</tr>
<tr>
<td>WeChat</td>
<td>806</td>
</tr>
<tr>
<td>QZone</td>
<td>652</td>
</tr>
<tr>
<td>Tumblr</td>
<td>555</td>
</tr>
<tr>
<td>Instagram</td>
<td>500</td>
</tr>
<tr>
<td>Twitter</td>
<td>313</td>
</tr>
</tbody>
</table>

"One trillion edges: graph processing at Facebook-scale."
Ching et al., VLDB 2015
Computational bottlenecks

**In theory:**

**Space**

\[ \mathcal{O}(m), \mathcal{O}(n^2) \] to store

**Time**

\[ \mathcal{O}(n^2) \] to construct

\[ \mathcal{O}(n^3) \] to run algorithms

**In practice:**

- 2012 Common Crawl Corpus:
  3.5 Billion pages (45 GB)
  128 Billion edges (331 GB)

- Pagerank on Facebook Graph:
  3 minutes per iteration, hundreds of iterations, tens of hours
  on 200 machines, run once per day
Two phases

1. Preprocessing:
   - **From vectorial data:** Collect a dataset $\mathbf{X} \in \mathbb{R}^{n \times d}$, construct a graph $\mathbf{G}$ using a similarity function.
   - **Prepare the graph:** Need to check if graph is connected, make it directed/undirected, build Laplacian.
   - **Load it on the machine:** On a single machine if possible, if not find smart way to distribute it.

2. Run your algorithm on the graph.
Large scale graph construction

Main bottleneck: \textbf{time}

- Constructing $k$-nn graph takes $O(n^2 \log(n))$, too slow
- Constructing $\varepsilon$ graph takes $O(n^2)$, still too slow
- In both cases bottleneck is the same, given a node finding close nodes ($k$ neighbours or $\varepsilon$ neighbourhood)

Fundamental limit: just looking at all similarities already too slow.

Can we find close neighbours without checking all distances?
Distance Approximation

Split your data in small subset of close points

Can find efficiently some (not all) of the neighbours.

- Iterative Quantization
- KD-Trees
- Locality Sensitive Hashing (LHS)

More general problem: learning good codeword representation
Storing graph in memory

Main bottleneck: **space**.
As a Fermi (back-of-the-envelope) problem

- Storing a graph with \( m \) edges require to store \( m \) tuples \((i, j, w_{i,j})\) of 64 bit (8 bytes) doubles or int.
- For standard cloud providers, the largest compute-optimized instances has 36 cores, but only 60 GB of memory.
- We can store \( 60 \times 1024^3 / (3 \times 8) \sim 2.6 \times 10^9 \) (2.6 billion) edges in a single machine memory.
Storing graph in memory

But wait a minute

- Natural graphs are sparse.
  - For some it is true, for some it is false (e.g. Facebook average user has 300 friends, Twitter averages 208 followers)
  - Subcomponents are very dense, and they grow denser over time

- I will construct my graph sparse
  - Losing large scale relationship, losing regularization

- I will split my graph across multiple machines
  - Your algorithm does not know that.
    - What if it needs nonlocal data? Iterative algorithms?
    - More on this later
Graph Sparsification

**Goal**: Get graph $G$ and find sparse $H$
Graph Sparsification: What is sparse?

What does \textbf{sparse} graph mean?

- average degree \(< 10\) is pretty sparse
- for billion nodes even \(100\) should be ok
- in general: average degree \(< \text{polylog } n\)

Are all edges important?

in a tree — sure, in a dense graph perhaps not
Graph Sparsification: What is good sparse?

Good sparse by Benczúr and Karger (1996) = cut preserving!

\[ H \approximates G \text{ well iff } \forall S \subseteq V, \text{ sum of edges on } \delta S \text{ remains } \]

\[ (1 \pm \epsilon) \]

\[ \delta S = \text{edges leaving } S \]

https://math.berkeley.edu/~nikhil/
Graph Sparsification: What is good sparse?

Good sparse by Benczúr and Karger (1996) = cut preserving!

Why did they care? faster mincut/maxflow

Recall what is a cut: \[ \text{cut}_G(S) = \sum_{i \in S, j \in \overline{S}} w_{i,j} \]

Define \( G \) and \( H \) are (1 ± ε)-cut similar when \( \forall S \)

\[ (1 - \varepsilon) \text{cut}_H(S) \leq \text{cut}_G(S) \leq (1 + \varepsilon) \text{cut}_H(S) \]

Is this always possible? Benczúr and Karger (1996): Yes!

\[ \forall \varepsilon \exists (1 + \varepsilon)-\text{cut similar } \tilde{G} \text{ with } O(n \log n/\varepsilon^2) \text{ edges s.t. } E_H \subseteq E \]

and computable in \( O(m \log^3 n + m \log n/\varepsilon^2) \) time \( n \) nodes, \( m \) edges
Graph Sparsification: What is good sparse?

\[ G = K_n \]

\[ H = d\text{-regular} \text{ (random)} \]

How many edges?

\[ |E_G| = \mathcal{O}(n^2) \]

\[ |E_H| = \mathcal{O}(dn) \]
Graph Sparsification: What is good sparse?

\( G = K_n \)

\( H = d\text{-regular (random)} \)

What are the cut weights for any \( S \)?

\[
\begin{align*}
w_G(\delta S) &= |S| \cdot |\overline{S}| \\
w_H(\delta S) &\approx \frac{d}{n} \cdot |S| \cdot |\overline{S}|
\end{align*}
\]

\[\forall S \subset V : \frac{w_G(\delta S)}{w_H(\delta S)} \approx \frac{n}{d}\]

Could be large :(

What to do?
Graph Sparsification: What is **good** sparse?

$G = K_n$

$H = d$-regular (random)

What are the cut weights for any $S$?

$$w_G(\delta S) = |S| \cdot |\bar{S}|$$

$$w_H(\delta S) \approx \frac{d}{n} \cdot \frac{n}{d} \cdot |S| \cdot |\bar{S}|$$

$$\forall S \subset V : \frac{w_G(\delta S)}{w_H(\delta S)} \approx 1$$

Benczúr & Karger: Can find such $H$ quickly for any $G$!
Graph Sparsification: What is good sparse?

Recall if $f \in \{0, 1\}^n$ represents $S$ then $f^T L_G f = \text{cut}_G(S)$

$$(1 - \varepsilon)\text{cut}_H(S) \leq \text{cut}_G(S) \leq (1 + \varepsilon)\text{cut}_H(S)$$

becomes

$$(1 - \varepsilon)f^T L_H f \leq f^T L_G f \leq (1 + \varepsilon)f^T L_H f$$

If we ask this only for $f \in \{0, 1\}^n \rightarrow (1 + \varepsilon)$-cut similar \cite{BenczurKarger1996}

If we ask this for all $f \in \mathbb{R}^n \rightarrow (1 + \varepsilon)$-spectrally similar \cite{SpielmanTeng2004}

Spectral sparsifiers are stronger!

but checking for spectral similarity is easier
Spectral Graph Sparsification

Rayleigh-Ritz gives:
\[ \lambda_{\text{min}} = \min \frac{x^T L x}{x^T x} \quad \text{and} \quad \lambda_{\text{max}} = \max \frac{x^T L x}{x^T x} \]

What can we say about \( \lambda_i(G) \) and \( \lambda_i(H) \)?

\[ (1 - \varepsilon) f^T L_G f \leq f^T L_H f \leq (1 + \varepsilon) f^T L_G f \]

Eigenvalues are approximated well!

\[ (1 - \varepsilon) \lambda_i(G) \leq \lambda_i(H) \leq (1 + \varepsilon) \lambda_i(G) \]

Using matrix ordering notation \((1 - \varepsilon) L_G \preceq L_H \preceq (1 + \varepsilon) L_G\)

As a consequence, \( \arg \min_x \| L_H x - b \| \approx \arg \min_x \| L_G x - b \| \)
Spectral Graph Sparsification

Let us consider unweighted graphs: $w_{ij} \in \{0, 1\}$

\[
L_G = \sum_{ij} w_{ij} L_{ij} = \sum_{ij \in E} L_{ij} = \sum_{ij \in E} (\delta_i - \delta_j)(\delta_i - \delta_j)^T = \sum_{e \in E} b_e b_e^T
\]

We look for a subgraph $H$

\[
L_H = \sum_{e \in E} s_e b_e b_e^T \quad \text{where } s_e \text{ is a new weight of edge } e
\]

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification

We want \((1 - \varepsilon)L_G \preceq L_H \preceq (1 + \varepsilon)L_G\)

Equivalent, given \(L_G = \sum_{e \in E} b_e b_e^T\) find \(s\), s.t. \(L_G \preceq \sum_{e \in E} s_e b_e b_e^T \preceq \kappa \cdot L_G\)

Forget \(L\), given \(A = \sum_{e \in E} a_e a_e^T\) find \(s\), s.t. \(A \preceq \sum_{e \in E} s_e a_e a_e^T \preceq \kappa \cdot A\)

Same as, given \(I = \sum_{e \in E} v_e v_e^T\) find \(s\), s.t. \(I \preceq \sum_{e \in E} s_e v e v_e^T \preceq \kappa \cdot I\)

How to get it? \(v_e \leftarrow A^{-1/2} a_e\)

Then \(\sum_{e \in E} s_e v_e v_e^T \approx I \iff \sum_{e \in E} s_e a_e a_e^T \approx A\)

multiplying by \(A^{1/2}\) on both sides
Spectral Graph Sparsification: Intuition

How does $\sum_{e \in E} v_e v_e^T = I$ look like geometrically?

Decomposition of identity: $\forall \mathbf{u}$ (unit vector): $\sum_{e \in E} (\mathbf{u}^T v_e)^2 = 1$

moment ellipse is a sphere

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification: Intuition

What are we doing by choosing H?

We take a subset of these $e_e$s and scale them!

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification: Intuition

What kind of scaling do we want?

Such that the blue ellipsoid looks like identity!

the blue eigenvalues are between 1 and $\kappa$

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification: Intuition

Example: What happens with $K_n$?

$K_n$ graph

$$\sum_{e \in E} b_e b_e^T = L_G$$

$$\sum_{e \in E} v_e v_e^T = I$$

It is already isotropic! (looks like a sphere)

rescaling $v_e = L^{-1/2} b_e$ does not change the shape

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification: Intuition

Example: What happens with a dumbbell?

\[ K_n \text{ graph} \]

\[ \sum_{e \in E} b_e b_e^T = L_G \]

\[ \sum_{e \in E} v_e v_e^T = I \]

The vector corresponding to the link gets stretched!

because this transformation makes all the directions important

rescaling reveals the vectors that are critical

https://math.berkeley.edu/~nikhil/
Spectral Graph Sparsification: Intuition

What is this rescaling $v_e = L_G^{-1/2}b_e$ doing to the norm?

$$\|v_e\|^2 = \|L_G^{-1/2}b_e\|^2 = b_e^T L_G^{-1} b_e = R_{eff}(e)$$

Reminder $R_{eff}(e)$ is the potential difference between the nodes when injecting a unit current.

In other words: $R_{eff}(e)$ is related to the edge importance!

**Electrical intuition:** We want to find an electrically similar $H$ and the importance of the edge is its effective resistance $R_{eff}(e)$.

Edges with higher $R_{eff}$ are more **electrically significant**!
Spectral Graph Sparsification

Todo: Given \( I = \sum_e v_e v_e^T \), find a sparse reweighting.

Randomized algorithm that finds \( s \):

- Sample \( n \log n/\epsilon^2 \) with replacement \( p_i \propto \|v_e\|^2 \) (resistances)
- Reweigh: \( s_i = 1/p_i \) (to be unbiased)

Does this work?

Application of Matrix Chernoff Bound by Rudelson (1999)

\[
1 - \epsilon \prec \lambda \left( \sum_e s_e v_e v_e^T \right) \prec 1 + \epsilon
\]

finer bounds now available

What is the biggest problem here? Getting the \( p_i \)s!
Spectral Graph Sparsification

We want to make this algorithm fast. How can we compute the effective resistances?

Solve a linear system $\hat{x} = \arg\min_x \|L_G x - b_e\|$ and then $R_{\text{eff}} = b_e^T \hat{x}$

- Gaussian Elimination: $O(n^3)$
- Fast Matrix Multiplication: $O(n^{2.37})$
- Spielman & Teng (2004): $O(m \log^{30} n)$
- Koutis, Miller, and Peng (2010): $O(m \log n)$

Fast solvers for SDD systems:
- use sparsification internally all the way until you hit the turtles
- still unfeasible when $m$ is large
Spectral Graph Sparsification

Chicken and egg problem

We need $R_{\text{eff}}$ to compute a sparsifier $H$ \leftarrow \\
\downarrow \text{We need a sparsifier } H \text{ to compute } R_{\text{eff}}

Sampling according to approximate effective resistances

$R_{\text{eff}} \leq \tilde{R}_{\text{eff}} \leq \alpha R_{\text{eff}}$ give approximate sparsifier $L_{G} \preceq L_{H} \preceq \alpha \kappa L_{G}$

Start with very poor approximation $\tilde{R}_{\text{eff}}$ and poor sparsifier.

Use $\tilde{R}_{\text{eff}}$ to compute an improved approximate sparsifier $H$ \leftarrow \\
\downarrow \text{Use the sparsifier } H \text{ to compute improved approximate } \tilde{R}_{\text{eff}}

Computing $\tilde{R}_{\text{eff}}$ using the sparsifier is fast ($m = O(n \log(n))$), and not too many iterations are necessary.
What can I use sparsifiers for?

- Graph linear systems: minimum cut, maximum flow, Laplacian regression, SSL
- More in general, solving Strongly Diagonally Dominant (SDD) linear systems
  - electric circuit, fluid equations, finite elements methods
- Various embeddings: k-means, spectral clustering.

But what if my problems have no use for spectral guarantees?

Or if my boss does not trust approximation methods
Distributed graph processing

Large graphs do not fit in memory

Get more memory
  ▪ Either slower but larger memory
    Or fast memory but divided among many machines

Many challenges
  Needs to be scalable
    ▪ minimize pass over data / communication costs
  Needs to be consistent
    ▪ updates should propagate properly
Distributed graph processing

Different choices have different impacts: for example splitting the graph according to nodes or according to edges.

Many computation models (academic and commercial) each with its pros and cons

MapReduce
MPI
Pregel
Graphlab
The GraphLab abstraction

Scope $S_1$

Edge Data
Vertex Data

$D_1 \leftrightarrow 2$
$D_2 \leftrightarrow 3$
$D_3 \leftrightarrow 4$
$D_3 \leftrightarrow 5$
$D_4 \leftrightarrow 5$
$D_5$

1
2
3
4
5
The GraphLab abstraction

In [1]: import sframe

In [2]: edges = sframe.SFrame.read_csv('/media/sf_share/td3_example_edges.csv')

In [3]: vertices = sframe.SFrame.read_csv('/media/sf_share/td3_example_vertices.csv')

In [4]: G = sframe.SGraph(edges=edges, vertices=vertices, src_field='src', dst_field='dst')

In [5]: G

Out[5]: SGraph({'num_edges': 26, 'num_vertices': 9})
   Vertex Fields:['__id', 'f']
   Edge Fields:['__src_id', '__dst_id', 'weight']
The GraphLab abstraction

Under the hood: tabular representation

Columns:
- __id int
- f float

Rows: 9

Data:

<table>
<thead>
<tr>
<th>__id</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.51</td>
</tr>
<tr>
<td>7</td>
<td>0.82</td>
</tr>
<tr>
<td>10</td>
<td>0.08</td>
</tr>
<tr>
<td>2</td>
<td>0.82</td>
</tr>
<tr>
<td>6</td>
<td>0.85</td>
</tr>
<tr>
<td>9</td>
<td>0.83</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
</tr>
<tr>
<td>1</td>
<td>0.35</td>
</tr>
<tr>
<td>4</td>
<td>0.36</td>
</tr>
</tbody>
</table>

[9 rows x 2 columns]

Columns:
- __src_id int
- __dst_id int
- weight float

Rows: 26

Data:

<table>
<thead>
<tr>
<th>__src_id</th>
<th>__dst_id</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>5</td>
<td>0.13185</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0.13185</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0.026779</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>0.57121</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>0.57121</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0.94047</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>0.94047</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>0.64528</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.93374</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>0.31713</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.57796</td>
</tr>
</tbody>
</table>

[26 rows x 3 columns]

Note: Only the head of the SFrame is printed.
The GraphLab abstraction

In [1]: import sframe

In [2]: G = sframe.SGraph()

In [3]: G

Out[3]: SGraph({'num_edges': 0, 'num_vertices': 0})
    Vertex Fields:{'__id']
    Edge Fields:{'__src_id', '__dst_id']

In [1]: import sframe

In [2]: G = sframe.SGraph()

In [3]: G

Out[3]: SGraph({'num_edges': 0, 'num_vertices': 0})
    Vertex Fields:{'__id']
    Edge Fields:{'__src_id', '__dst_id']

In [4]: G.add_edges(sframe.Edge(1,2))

Out[4]: SGraph({'num_edges': 1, 'num_vertices': 2})
    Vertex Fields:{'__id']
The GraphLab abstraction

- The graph is immutable. **why?**
- All computations are executed asynchronously
  - We do not know the order of execution
  - We do not even know where the node is stored
    - **what data can we access?**
- The data is stored in the graph itself
  - Only access **local data**
- Functional programming approach
The GraphLab abstraction

```python
triple_apply(triple_apply_fn, mutated_fields, input_fields=None)
```

processes all edges asynchronously and in parallel

```c
>>> PARALLEL FOR (source, edge, target) AS triple in G:
...  LOCK (triple.source, triple.target)
...  (source, edge, target) = triple_apply_fn(triple)
...  UNLOCK (triple.source, triple.target)
...  END PARALLEL FOR
```

- No guarantees on order of execution
- Updating (src,edge,dst) violates immutability
- `triple_apply_fn` receives a copy of (src,edge,dst)
  - returns an updated (src’,edge’,dst’)
  - use return values to build a new graph
The GraphLab abstraction

triple_apply_fn is a pure function

Function in the mathematical sense, same input gives same output.

```python
def triple_apply_fn(src, edge, dst):
    # can only access data stored in src, edge, and dst,
    # three dictionaries containing a copy of the
    # fields indicated in mutated_fields
    f = dst['f']

    # inputs are copies, this does not change original edge
    edge['weight'] = g(f)

    return ({'f': dst['f']}, edge, dst)
```
An example, computing degree of nodes

```python
def degree_count_fn (src, edge, dst):
    src[‘degree’] += 1
    dst[‘degree’] += 1
    return (src, edge, dst)

G_count = G.triple_apply(degree_count_fn, ‘degree’)
```
The GraphLab abstraction

Slightly more complicated example, suboptimal pagerank

```python
#assume each node in G has a field 'degree' and 'pagerank'
#initialize 'pagerank' = 1/n for all nodes

def weight_count_fn (src, edge, dst):
    dst['degree'] += edge['weight']
    return (src, edge, dst)

def pagerank_step_fn (src, edge, dst):
    dst['pagerank'] += (edge['weight'] * src['pagerank'] / dst['degree'])
    return (src, edge, dst)

G_pagerank = G.triple_apply(weight_count_fn, 'degree')

while not converged(G_pagerank):
    G_pagerank = G_pagerank.triple_apply(pagerank_step_fn, 'pagerank')
```

How many iterations to convergence?