Codes, Embeddings, & Non-Euclidean Geometry
Unexpected Allies in the Fight to Clean Data

Fred Sala
Three Tools

We’ll connect data cleaning to

• **Snorkel/Metal**: a weak supervision framework based on graphical models
  • Models and integrates user-provided data cleaning functions

• **Non-Euclidean Embeddings**: high-quality data representations
  • Efficiently converts data to form suitable for machine learning algorithms

• **Synch**: an error-correction coding approach to synchronization
  • Efficient data synchronization and deduplication for large files
Three Tools

We’ll connect data cleaning to

• **Snorkel/Metal**: a weak supervision framework based on graphical models
  • Models and integrates user-provided data cleaning functions

• **Non-Euclidean Embeddings**: high-quality data representations
  • Efficiently converts data to form suitable for machine learning algorithms

• **Synch**: an error-correction coding approach to synchronization
  • Efficient data synchronization and deduplication for large files
1. Snorkel + Metal: A System for Rapidly Creating Training Sets


Goal: Bring all sources to bear to program ML systems in a radically faster and easier way
The Snorkel Pipeline

1. Users write *labeling functions* to generate noisy labels
2. Snorkel models and combine these labels
3. We use the resulting *probabilistic* training labels to train a model

Key point: Input is *labeling functions* — **No hand-labeled training sets**
Goal: Classify Events in Medical Reports

• Challenges:
  
  • Medical data extremely expensive to label
  
  • ...And labeling schema changes with institution, setting, machine, etc.
  
  • Diverse, multi-modal data

“Indication: Chest pain. Findings: Focal consolidation and pneumothorax...”
Step 1: Writing Labeling Functions

Users write \textit{labeling functions} to generate noisy labels.
Simple LF Example: Pattern Matching


```
def LF_pneumothorax(c):
    if re.search(r'pneumo.*', c.report.text):
        return "ABNORMAL"
```

Label = ABNORMAL

*Labeling functions (LFS)* are black box functions that can express domain expertise
Simple LF Example: Pattern Matching

“Indication: Chest pain. Findings: No focal consolidation or pneumothorax...”

However, LFs can be noisy! We can estimate their accuracies to handle this.
Step 2: Model, Combine, Iterate

1. Users write *labeling functions* to generate noisy labels
2. Snorkel models and combine these labels
Step 3: Train End Model

1. Users write labeling functions to generate noisy labels
2. Snorkel models and combine these labels
3. We use the resulting probabilistic training labels to train a model
We Can Now Train A Commodity Model To...

- **Generalize beyond the LFs**
  
  "Indication: Chest pain. Findings: Collapsed lung."

- **Transfer to new modalities or edge deployment settings**

  Increases recall by 43% on average - up to 24 pts.! [VLDB’18]

End result: within several pts. of *years* of hand-labeling in 1 *week*!
How Do We Model and Combine LFs?

Core Technical Challenge: How to best reweight and combine the noisy supervision signal?
A Generative Model of the Training Data Labeling Process

Core Technical Challenge: How to estimate these parameters without any ground-truth labels?
A Generative Model of the Training Data Labeling Process

\[ Y_i \in \{\text{NORMAL, ABNORMAL}\} \]

**Learned Accuracies**
- LF 1: 90%
- LF 2: 80%
- LF 3: 60%

**Probabilistic Training Labels**

Intuition: Learn the accuracies from the *overlaps*
Modeling Correlated LFs is Crucial

We can learn dependency structure using statistical and/or static analysis techniques [ICML ’17, NIPS ‘17]
Matrix Completion: Underlying Problem

\[
\mathbf{O} = \begin{bmatrix}
1.0 & 0.8 & -0.1 \\
0.8 & 1.0 & 0.2 \\
-0.1 & 0.2 & 1.0
\end{bmatrix}
\]

\[
\underset{\mathbf{z}}{\text{argmin}} \| \mathbf{O}^{-1} - \mathbf{z} \mathbf{z}^T \|_\Omega
\]

Fit directly to the observed overlaps --> matrix completion problem!
Takeaways

- **Data cleaning techniques** can be used as LFs
- Automatically synthesize and denoise!
- Related idea developed by folks (formerly) in our lab:


- **Snorkel**: improves **scalability**!
  - Matrix-completion approach much faster, scales up!
Three Tools

We’ll connect data cleaning to

• **Snorkel/Metal**: a weak supervision framework based on graphical models
  • Models and integrates user-provided data cleaning functions

• **Non-Euclidean Embeddings**: high-quality data representations
  • Efficiently converts data to form suitable for machine learning algorithms

• **Synch**: an error-correction coding approach to synchronization
  • Efficient data synchronization and deduplication for large files
2. Embeddings & Non-Euclidean Geometry

- Idea: map discrete objects (words, people, databases, media items) into continuous spaces + preserve relationships

- Enables using objects with modern ML learning
  - Clean data with ML tools?

- Enable math operations:
  - “woman” - “man” = “queen” and “king”
Preserving Relationships

• Critical: preserving relationships between objects
• These relationships can be viewed as a graph

• Hierarchical relationships: trees
  • Ex: Artists -> Albums -> Songs

• How well can we embed trees in traditional Euclidean space?
Geometry Matters!

• **Q:** How well can we embed trees in Euclidean space?
  • **A:** Not very well! We can’t do it perfectly with *any number of dimensions!*  
  • Via a sophisticated result (Bourgain’s theorem)

• **Why?**
  • We want to preserve the *graph distances* (shortest paths)
  • From 5 to 7 -> go through 6: *two hops*
  • But in the plane, 5 and 7 are much closer
  • Need *lots of dimensions* just to get close!
Hyperbolic Embeddings?

• Embeddings typically use Euclidean space
  • Poor choice for hierarchical data (i.e., trees): implicitly noisy representations!

• Instead, embed into hyperbolic space (non-Euclidean)
  • As we will see, a natural fit for trees

• How do we work with this strange space?
  • Challenges: efficient embedding algorithms, scaling, limits, special operations

Hyperbolic Space

• A special non-Euclidean space works to embed trees
  • With even just 2 dimensions!

• **Hyperbolic space**: Interesting and useful geometry
  • Negative curvature
    • more “room” vs. Euclidean space

• Ideal space for hierarchical data

*Escher, Circle Limit IV*
Hyperbolic Embeddings: The **Promise**

- **Artist/album/song hierarchy**

<table>
<thead>
<tr>
<th>Song 1</th>
<th>Song 2</th>
<th>Bands</th>
<th>Albums</th>
<th>Distance</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>With a Little Help From My Friends</td>
<td>Lucy in the Sky With Diamonds</td>
<td>Beatles, Beatles</td>
<td><img src="image1.png" alt="Image" /> <img src="image2.png" alt="Image" /></td>
<td>3.761</td>
<td>Same album and artist</td>
</tr>
<tr>
<td>With a Little Help From My Friends</td>
<td>While My Guitar Gently Weeps</td>
<td>Beatles, Beatles</td>
<td><img src="image3.png" alt="Image" /> <img src="image4.png" alt="Image" /></td>
<td>4.823</td>
<td>Same artist, different album</td>
</tr>
<tr>
<td>With a Little Help From My Friends</td>
<td>Back in Black</td>
<td>Beatles, AC-DC</td>
<td><img src="image5.png" alt="Image" /> <img src="image6.png" alt="Image" /></td>
<td>23.430</td>
<td>Unrelated</td>
</tr>
</tbody>
</table>

- **Generic way to embed hierarchical structures**
- Can use **for cleaning too**
- Lots of **challenges!**
Tree Embedding Algorithm

- Powerful tool for embedding trees
  - Recursive approach without optimization
  - Achieves arbitrarily good distortion

- For each embedded node, place children into disjoint subcones
  - If $d_H(parent, child)$ is long enough, cones are always disjoint
  - We scale all distances by a constant factor, divide by it when reconstructing

- Takeaway: Scaling matters!
Scaling and Subtree Separation

- Example with 7-ary tree
- **Cone separation** is a function of scaling
  - Small distances: not enough separation
  - Separation improves with scaling
More!

- Powerful tool for theoretical results
  - Reveals a precision/quality/dimension tradeoff

- Great results in practice!
  - Distortion (lower is better!)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Nodes, #Edges</th>
<th>Type</th>
<th>Euclidean 100 Dim Distortion</th>
<th>Hyperbolic 2 Dim Distortion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phylogenetic Tree Tree</td>
<td>344, 343</td>
<td>Tree</td>
<td>0.746</td>
<td>0.006</td>
</tr>
<tr>
<td>CS Ph.D. Advisor Network</td>
<td>1025, 1043</td>
<td>Tree-like</td>
<td>0.708</td>
<td>0.286</td>
</tr>
<tr>
<td>Gr-QC ArXiv Dense</td>
<td>4158, 13428</td>
<td>Dense</td>
<td>0.546</td>
<td>0.354</td>
</tr>
</tbody>
</table>
PyTorch Visualizations

• Optimizer: 20 node *cycle* and ternary *tree*
Takeaways

• We can embed everything...

• But we have to be careful: the structure of data needs to match the geometry of the embedding space

• Otherwise, our representations are noisy!

• Hyperbolic space: the right choice for hierarchical data
Three Tools

We’ll connect data cleaning to

• **Snorkel/Metal**: a weak supervision framework based on graphical models
  • Models and integrates user-provided data cleaning functions

• **Non-Euclidean Embeddings**: high-quality data representations
  • Efficiently converts data to form suitable for machine learning algorithms

• **Synch**: an error-correction coding approach to synchronization
  • Efficient data synchronization and deduplication for large files
3. Synchronization via Codes

• **Synchronize** or **Deduplicate** Large Files

• Idea: exploit simple **error-correcting code**
  • Corrects **fixed** number of edits

• Divide-and-Conquer. Result: efficient algorithm + theoretical guarantees.

**FS, C. Schoeny, N. Bitouzé, L. Dolecek, “Synchronizing files from a large number of insertions and deletions,” TCOM 2016.**
Synchronization: Start With a File...

Original File

```c
int main() {
  char *filename = "example.txt";
  // Open file
  int fd = open(filename, O_RDONLY);
  if (fd == -1) {
    // Handle error
  }
  // Read file
  char buffer[1024];
  while (read(fd, buffer, sizeof(buffer)) > 0) {
    // Process buffer
  }
  // Close file
  close(fd);
  return 0;
}
```
Synchronization: Edit it...

Original File

Alice's Version

Edits

Edits

Bob's Version
Synchronize the Changes!
Done!

Synchronization Protocol
so that Bob’s version matches Alice’s
Tour of the Algorithm

Node A

File X: hdcatgbkujinntuqjrvxfqxrzleydwajgnhtwanyohdcyhzrnrm

Node B

File Y: hdcatbkujintkuqjrvxqxrledwrajgnidtwayohdcyhzrnrm
Split into Pivot & Segment Strings

Node A
Pivot Strings: short
Segment Strings: long

File X: hdcatgbkujintuqjrvxfqxrzleydwajgndtwanyohdcyhzrnrm

File Y: hdcatbkujintkuqjrvqxqxrledwrajgnidtwayohdocyhzrnrm

Node B
Send Pivots

Node A

Pivot Strings: short
Segment Strings: long

File X: hdcatgbkjlnrjtyfqrzldyewjndtwanyohdcyhzrnm

Send the pivots:
hdcrvgnrdnm

File Y: hdcatbkjlnrtqjrvxqxrledwrajgnidtwahdydzrhm

Node B
Match Pivots

Node A

- **Pivot Strings**: short
- **Segment Strings**: long

File X: hdcatgbkujintuqjrvxfqxrzleydwajgndtwanyohdcyhznrm

Send the pivots:

- hdc jrv gnd nrm

File Y: hdcatbkujintkuqjrvxqxrledwrajgnidtwayohdcyhznrm

Node B

- **Matched Pivots**
Sometimes Pivots Aren’t Matched...

**Node A**
- **Pivot Strings**: short
- **Segment Strings**: long

File X: hdcatgbkujinntuqjrvxfqxrzydwayjgnjtwanyohdcyhzrnrm

**Node B**
- **Matched Pivots**

File Y: hdcatbkujintkuqjrvxqxrledwrajgnidtwayohdcyhzrnrm
Segments are Fixed-Length
Synchronize Via **ECC**!

**Node A**

- **Pivot Strings**: short
- **Segment Strings**: long

**File X:**

```
hdcatgbkujinntuqjrvxfqxrzleydwajgndtwanyohdcyhzrnr
```

**Node B**

- **Matched Pivots**
- **Non-Synced Segments**

**File Y:**

```
hdcatbkujintkuqjrvxqxrledwrajgnidtwayohdcyhzrnr
```
Done

Node A

- **Pivot Strings**: short
- **Segment Strings**: long

File X: hdcatgbkujinntuqjrvxfqxrzleydwajgndtwanyohdcyhzrnrnrm

File Y: hdcatgbkujinntuqjrvxfqxrzleydwajgndtwanyohdcyhzrnrnrm

Node B

- **Matched Pivots**
- **Synchronized Segments**
Matching the Pivots

Any pivot might match **multiple times**. How do we choose? Weighted graph formulation: pick heavy path
Results

\( n = 50000 \), i.i.d. file, i.i.d. edits, \( q = 52 \),
Our pivot length: 5, segment length: \( 1/\beta \).

\[ \text{x-axis: edit rate. Comparison against rsync tool.} \]
Takeaways

- **Coding theory**: really efficient way to synchronize (and fix lots of kinds of noise)
  - Sometimes optimal!

- But only for some settings...
  - Fixed length string, fixed number of edits

- A little bit of work gets us an efficient + general approach
  - Can we apply similar coding-theory approaches elsewhere?
Thank you!

• Our tools have open source implementations available on Github!

• We’d love to hear your feedback and ideas:

fredsala@stanford.edu