CS435 BIG DATA

PART 1.
LARGE SCALE DATA ANALYSIS USING MAPREDUCE

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics

- FAQs
- Shuffle and Sort
- Finding similar items at scale

FAQs

- Demo for PA1 is going on

Shuffle and Sort

- Sort
  - MapReduce makes the guarantee that the input to every reducer is sorted by key
- Shuffle
  - MapReduce transfers the map outputs to the reducers as inputs
**Shuffle and sort: In a map task**

- Each map task has a circular memory buffer
  - For output
    - 100MB by default
  - `io.sort.mb`

- When the contents of the buffer reaches the threshold, a background thread starts spill the contents to disk
  - Default 0.80
  - Spills are written in round-robin-fashion to the local directory
    - `mapred.local.dir`

**Partitioning**

- Before the data is written to the local disk, data is divided into partitions corresponding to the reducers

- The background thread performs an in-memory sort by key
  - Within each partition

- Each time the memory buffer reaches the spill threshold, a new spill file is created
  - There can be several spill files after the last output record is written
  - The spill files are merged into a single partitioned and sorted output file

**Output file**

- Combiner is run
- Output files from map can be compressed

- The output file’s partitions are made available to the reducers over HTTP

**Copy phase**

- The reduce task needs the map output from several map tasks across the cluster

- Copy phase
  - The reduce task starts copying their outputs as soon as each completes
  - The map tasks may finish at different times
  - Merges them into larger and sorted files
  - Decompresses the compressed files
Sort phase

- Sort phase
  - All of the map outputs should be moved and copied to the reduce task
  - Merging and sorting the map outputs
  - Sorting is done in rounds
    - If there are 50 map outputs and the merge factor was 10
      - 5 intermediate files
      - Merging intermediate files: additional round
      - 6 rounds will be required
    - Final round
      - A mixture of in-memory and on-disk segments
      - Directly feeds the reduce function
      - Without write a single sorted file to disk

MapReduce

MapReduce is inspired by the concepts of map and reduce in Lisp.

Developed within Google as a mechanism for processing large amounts of raw data
- Crawled documents or web request logs
- Distributes those data across thousands of machines
- Same computations are performed on each CPU with different dataset

Part 1.
Large scale data analysis using mapreduce
3. Finding similar items at scale

Similarity of Documents

- Finding textually similar documents in a large corpus
  - Examples
    - Plagiarism
    - Mirror pages
      - Articles from the same source
      - News aggregators
    - Tries to find all versions of articles from the same source to show only one
  - Collaborative Filtering
    - Amazon's item-to-item CF
    - Movie recommendation

This material is developed based on,
Jaccard Similarity

Jaccard Coefficient (without description)
- Compare two sets $P$ and $Q$ with the following formula:
  \[ \text{StringJaccard}(P, Q) = \frac{|P \cap Q|}{|P \cup Q|} \]
- Measures the fraction of the data that is shared between $P$ and $Q$
- Compared to all data available in the union of these two sets.
- What are $P$ and $Q$?
  - Set of tokens from Strings
  - Complete description about data (candidates)

Example
- DescriptionJaccard and StringJaccard have the same value. Is this always true?
  \[ \text{DescriptionJaccard}(c_1, c_2) = \frac{|OD(c_1) \cap OD(c_2)|}{|OD(c_1) \cup OD(c_2)|} \]
  \[ \text{StringJaccard}(P, Q) = \frac{|P \cap Q|}{|P \cup Q|} \]

- Now, specify the parts of a person’s name as title, firstname, middlename, and lastname
  \[ OD(c_1) = \{\text{firstname, Thomas}, \text{middlename, Sean}, \text{lastname, Connery}\} \]
  \[ OD(c_2) = \{\text{title, Sir}, \text{middlename, Sean}, \text{lastname, Connery}\} \]
  \[ \text{DescriptionJaccard}(c_1, c_2) = 2 / 4 = 0.5 \]
  \[ \text{StringJaccard}(P, Q) = 2 / 4 = 0.5 \]

Example
Example continued

What if “Sean” would have been put in the firstname/ middlename attribute?

OD(c₁) = {(middlename, Thomas), (firstname, Sean), (lastname, Connery)}
OD(c₂) = {(title, Sir), (middlename, Sean), (lastname, Connery)}

\[
\text{Description Jaccard} (c₁, c₂) = \frac{1}{5}
\]

Deficiencies of the Jaccard Similarity

• Some attribute is more descriptive
  • Title is less descriptive than firstname and lastname

• Very sensitive to typographical errors in single tokens
  • Shean Conery and Sean Connery have a similarity of zero.

Cosine similarity

• Given two \( n \)-dimensional vectors \( V \) and \( W \), the cosine similarity computes the cosine of the angle \( \alpha \) between these two vectors as

\[
\text{Cosine Similarity}(V, W) = \cos(\alpha) = \frac{V \cdot W}{\|V\| \cdot \|W\|}
\]

Where \( \|V\| \) is the length of the vector \( V = [a, b, c, \ldots] \) computed as

\[
\sqrt{a^2 + b^2 + c^2 + \ldots}
\]

Cosine similarity - Continued

• The vectors \( V \) and \( W \)
  • Tokens in a string
  • Descriptions of a candidate

• The \( d \) dimensions of these vectors correspond to all \( d \) distinct tokens in a set of strings.
  • Denoted as \( D \)

• For a large database, \( d \) may be large
  • \( V \) and \( W \) have high dimensionality \( d \)

Weight of Token

• Vector contains a weight for each of the \( d \) distinct tokens

• How to measure the weight?
  • Measuring frequency
  • Term frequency – inverse document frequency (tf-idf)
Inverse document frequency

- Assigns higher weights to tokens that occur less frequently in the scope of all candidate descriptions.

\[ \text{idf}_{t,c} = \log \frac{N}{|\{c | (a,v) \in OD(c) \wedge t \in \text{tokenize}(v)\}|} \]

for the total number of candidates, \(N\)

**tf-idf weighting**

- The product of its \(tf\) weight and its \(idf\) weight

\[ W_{t,d} = (1 + \log_{10} tf_{t,d}) \times \log_{10} \left(\frac{N}{df_t}\right) \]

For the total number of candidates, \(N\)

- Best known weighting scheme in information retrieval
  - Note: the "-" in \(tf-idf\) is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf \(\times\) idf
  - Increases with the number of occurrences within a document
  - Increases with the rarity of the term in the collection

**Example**

<table>
<thead>
<tr>
<th>CID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>Allstate</td>
</tr>
<tr>
<td>c2</td>
<td>American Automobile Association</td>
</tr>
<tr>
<td>c3</td>
<td>American National Insurance Company</td>
</tr>
<tr>
<td>c4</td>
<td>Farmers Insurance</td>
</tr>
<tr>
<td>c5</td>
<td>GEICO</td>
</tr>
<tr>
<td>c6</td>
<td>John Hancock Insurance</td>
</tr>
<tr>
<td>c7</td>
<td>Liberty Insurance</td>
</tr>
<tr>
<td>c8</td>
<td>Mutual Insurance of America Life Insurance</td>
</tr>
<tr>
<td>c9</td>
<td>Safeway Insurance Group</td>
</tr>
<tr>
<td>c10</td>
<td>Westfield</td>
</tr>
</tbody>
</table>

**Example continued**

- Compute the similarity between the two strings \(s1=\text{Farmers Insurance}\), \(s2=\text{Liberty Insurance}\)

\[ \text{idf}_{\text{Insurance}, c4} = \log_{10}(10/6) \]

\[ \text{tf-idf}_{\text{Farmers}, c4} = (1+\log_{10} 1) \times \log_{10}(10/1) = 1 \]

\[ \text{tf-idf}_{\text{Insurance}, c4} = (1+\log_{10} 1) \times \log_{10}(10/6) = 0.23 \]

\[ \text{tf-idf}_{\text{Liberty}, c7} = (1+\log_{10} 1) \times \log_{10}(10/1) = 1 \]

\[ \text{tf-idf}_{\text{Insurance}, c7} = (1+\log_{10} 1) \times \log_{10}(10/6) = 0.23 \]
Example continued

- Compute the similarity between the two strings $s_1=$Farmers Insurance, $s_2=$ Liberty Insurance

$$\text{CosSimilarity}(V,W) = \cos(\alpha) = \frac{V \cdot W}{||V|| \times ||W||} = \frac{0.23 \times 0.23}{\sqrt{0.23^2 + 1^2}} = 0.047$$

- What is the Jaccard similarity for the same case?

---

How can we compute TF-IDF based similarity analysis?

- Programming Assignment 2

Using n-Grams

A string is divided into smaller tokens of size $n$.
- $q$-grams or $n$-grams
- Size of $n$-gram is string with a length $n$
  - Size of $n$-gram words is string with a length $n$ words

Generating n-grams
- Slide a window of size $n$ over the string

Generating n-grams

$s_1 = \text{Henri Waternoose}$
$s_2 = \text{Henry Waternoose}$

- Generate 3-grams

- $Q$-grams of $s_1 = (\text{HH, HHe, Hen, enr, riri, rri, rri, _iW, _Wa, Wat, ate, ter, ern, mo, noo, oos, ose, seii, seifi})$
- $Q$-grams of $s_2 = (\text{HH, HHe, Hen, enr, rry, rry, _yW, _Wa, Wat, ate, ter, ern, mo, noo, ose, seii, seifi})$
n-gram based token similarity

- 13 overlaps among total 22 distinct n-grams.
- Overlaps: number of two item pairs having overlap

Jaccard similarity

\[
\text{StringJaccard}(s_1, s_2) = \frac{13}{22} = 0.59
\]

Using n-grams

- Using the same similarity measures used in other token based similarity computations
- Less sensitive to typographical errors
- What if we change the size of n?

How large should \( n \) be? [1/2]

- \( n \) should be picked large enough that the probability of any given n-gram appearing in any given document is low

Example

- Our corpus of documents is emails
  - Printable ASCII characters \( 2^5 = 14,348,907 \) possible 5-grams
  - Is it really true with real emails?

How large should \( n \) be? [2/2]

- All characters do not appear with equal probability
- Imagine that there are only 20 characters and estimate the number of n-grams as \( 20^n \)
- For research articles, a choice of \( n = 9 \) is considered safe
Near Neighbor (NN) Search
- Also known as proximity search, similarity search
- Finding closest or most similar points
- The post-office problem
- Assigning to a residence the nearest post office

Hashing n-grams
- Creating buckets and use the bucket numbers as the shingles
- For the set of 9-grams,
  - Map each of those 9-grams to a bucket number in the range 0 to $2^{32} - 1$
  - 9 bytes of data is compacted to 4 bytes
- Can hash-based approaches provide NN search results?

Applications of Near-neighbor search: Minhashing

Similarity-preserving summaries of set
- Signatures
  - Replacing large sets of n-grams by much smaller representations
  - We should be able to compare the signatures of two sets and estimate the Jaccard similarity
  - Of the underlying sets from the signatures alone

Matrix representations of sets

Minhashing
- Signature generating algorithm
  - Minhash of the characteristic matrix
  - Minhash of a set is the number of the row (element) with first non-zero in the permuted order $\pi$
  - $\pi = (b, e, a, d, c)$

<table>
<thead>
<tr>
<th>Element</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a, b, c, d, e)
S1 = (a, d)
S2 = (c)
S3 = (b, d, e)
S4 = (a, c, d)
Minhash and Jaccard Similarity

- **Theorem:**
  - \( P(\text{minhash}(S) = \text{minhash}(T)) = JaccardSIM(S,T) \)

**Proof:**

\[
\begin{align*}
X &= \text{number of rows with 1 for both } S \\
    & \quad \text{and } T \text{ (e.g. } x = 1) \\
Y &= \text{number of rows with either } S \text{ or } T \\
    & \quad \text{have 1, but not both (e.g. } y = 2) \\
Z &= \text{number of rows with both 0 (e.g. } z = 2) \\

P(\text{minhash}(S) = \text{minhash}(T)) &= \frac{X}{X+Y} = JaccardSIM(S,T)
\end{align*}
\]