Recursive SQL for Data Mining

Maximilian E. Schüle, Alfons Kemper, Thomas Neumann
Copenhagen, Denmark, July 7, 2022
In-Database Machine Learning: Problem

xkcd.org #2054 CC BY-NC 2.5
In-Database Machine Learning: Solution
In-Database Machine Learning

- SQL sufficient for machine learning (ML)
  - Turing-complete with recursive tables
- Idea
  - Data preprocessing using SQL
  - No need for data extraction out of a database system
Structure

Clustering
k-Means, DBSCAN

Graph Analysis
PageRank

Association Rules
Apriori
k-Means

```sql
SELECT longitude, latitude, clista
FROM
(SELECT p.longitude, p.latitude, clista AS llista,
  rank()
OVER (partition by p.lista)
ORDER BY (p.longitude-c.longitude)+(p.latitude-c.latitude)+(p.longitude-c.longitude)
OVER (partition by c.longitude) as c,
  (p.latitude-c.longitude) as l
FROM airports p,
  (SELECT *, rank()
  OVER (order by longitude, latitude) has cid
  FROM airports)
  name epk,
  gradient, cluster, labeling
  FROM HyperInsight
```
k-Means

- \( n \)-dimensional points \( x \in P \subset \mathbb{R}^n \)
- \( k \) clusters \( C \) with \( k \) points forming the initial centres \( C_0 \subset P, \lvert C_0 \rvert = k \).
- A point belongs to the closest located cluster \( c \in C \) based on a metric like Euclidean distance (\( ||c - x||_2 \))
- Eq. (1) returns all points of a cluster.
- new centre: average of all points (2):

\[
\text{cluster}(c) = \{ x \in P : \nexists d \in C : d \neq c \land ||d - x||_2 < ||c - x||_2 \},
\]

\[
c_{i+1} = \sum_{x \in \text{cluster}(c_i)} \frac{x}{\lvert \text{cluster}(c_i) \rvert}.
\]

- \( k \) tuples: centres (line 2)
- each iteration: update coordinates (line 4-10).
- window function: ranking of closest centres per point (line 5-86).

with recursive clusters (iter, cid, x, y) as 

( select 0, id, x, y from points limit 5 )

union all

select iter+1, cid, avg(px), avg(py) from ( 

select iter, cid, p.x as px, p.y as py, rank() over (partition by p.id order by 

(p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc, 

(c.x*c.x+c.y*c.y) asc) as 

(1) 

from points p, clusters c ) x 

where x.rank=1 and iter<100 group by cid, iter

) select * from clusters where iter=100;
DBSCAN

- $\varepsilon$: maximal distance between two points
- minimal number of points per cluster $\text{minPoints} > 1$, declared as noise otherwise.

For every point within a cluster $x \in G \subset P \subset \mathbb{R}^n$, another point $y \in G$ exists, whose distance to $x$ is less than $\varepsilon$:

$$\forall x \in G : \exists y \in G : x \neq y \land \| x - y \|_2 < \varepsilon.$$  (3)

- First, each point forms its own cluster (line 2).
- clusters that are less than $\varepsilon$ away are merged (line 4-8).

```
with recursive dbscan(iter, id, x, y, clusterid, noise) as
    ( select 0, id, x, y, id, true from points
      union all
      select iter+1, p.id, p.x, p.y, min(c.clusterid),
      count(*) < 3
      from points p, dbscan c
      where iter<10 and (p.x-c.x)^2+(p.y-c.y)^2<1.5^2
      group by iter, p.id, p.x, p.y
    )

select * from dbscan where iter=10;
```
PageRank

- importance of a web pages
- a link directing to another web page: an edge \((s, d) \in N \times N\).

1. each node: same PageRank value \(pr_0\) (4).
2. per iteration: each node \(s\) distributes its own value \(pr_i(s)\) equally to all outgoing edges \((s, d) \in E\).
3. new PageRank value \(pr_{i+1}(n)\) of a node \(n\): sum of the values of all incoming edges \((s, n) \in E\), damping factor \(\alpha\) (5):

\[
pr_0(n) := \frac{1}{|N|},
\]

\[
pr_{i+1}(n) := \alpha \cdot \sum_{(s, n) \in E} \frac{pr_i(s)}{| \{d | (s, d) \in E \} |} + \frac{1 - \alpha}{|N|}.
\]

1. base case (line 2) computes the initial PageRank value \(pr_0\).
2. recursive step: divides each node’s PageRank value by the number of outgoing edges (line 11-14), assigns and sums up this fraction for each destination node \(dst\) (line 9-18).

Maximilian E. Schüle (TUM) | Recursive SQL for Data Mining

with recursive pagerank (iter,node,pr) as ( 1
  select 0, e.dst, 1::float/( 2
    select count(distinct dst) from edges) 3
  from edges e 4
  group by e.dst 5
  union all 6
  select iter+1,dst,0.1*((1::float/ 7
    (select count(distinct dst) 8
    from edges)))+0.9*sum(b) 9
  from ( 10
    select iter, e.dst, p.pr/( 11
      select count (*) 12
      from edges x 13
      where x.src=e.src) as b 14
    from edges e, pagerank p 15
    where e.src=p.Node and iter < 100 16
  ) i 17
  group by dst, iter 18
) select * from pagerank where iter=100; 19
Apriori

with airline_rules AS
(SELECT * FROM apriori)
(SELECT airline_id,
destination_airport
FROM routes, apriori)
SELECT *
FROM airline_rules
WHERE "MUC" = ANY(airline_rules."Pre"​​);
Apriori

- Determining the frequent item sets for the Apriori algorithm
- A recursively growing relation calculates the frequent item sets: starting with the one-element item sets (each item as an array with one element).

```
WITH recursive transactions (tid, bucket) AS (  
  -- one array per shopping cart  
  SELECT tid, array_agg(item) FROM sales GROUP BY tid  
  -- frequent item sets of size 1
), sales_supp AS (  
  SELECT item FROM sales GROUP BY item HAVING COUNT(*) >= 10
), frequentitemsets AS (  
  -- frequent item sets with support >= 10  
  -- with one element
  (SELECT DISTINCT array[p.item]::INT[] AS items FROM sales_supp p)
  UNION ALL (  
    -- extend item sets recursively by one element
    SELECT DISTINCT array_append(t.items, p.item::INT)::INT[]
    FROM frequentitemsets t, sales_supp p
    WHERE 10 <= (  
      -- count support
      SELECT COUNT(*) FROM transactions t2
      WHERE array_append(t.items, p.item::INT)::INT[] @<@ (t2.bucket)
    ) AND t.items[((SELECT COUNT(*) FROM unnest(t.items))]<p.item
  ))
SELECT * FROM frequentitemsets;
```
Evaluation
Evaluation

- **HyPerScript**, table operators of HyPer and MADlib, recursive tables in HyPer, PostgreSQL and Umbra, and *PL/pgSQL* procedures (PostgreSQL 12.6 with MADlib 1.17.0)
- Ubuntu 20.04 LTS machine with six Intel Core i7-3930K CPUs running at 3.20 GHz and 64 GB DDR4 RAM
Evaluation: k-Means

- 10^6 points, x- and as y-coordinates equally distributed in [0, 10^6]
- runtime grows linearly with the input size
- integrated operators the best
- implementation within *HyPerScript*/recursive table comparable performance, outperform PostgreSQL.
- *HyPerScript*: 30 % faster with each additional core
Evaluation: DBSCAN

- $10^6$ points, x- and as y-coordinates equally distributed in $[0, 10^6]$
- recursive computation in Umbra as fast as the implemented operator

\[ \epsilon = 20, \ \text{minPts} = 2, \ 100 \ \text{iterations, 12 threads}. \]
Evaluation: Apriori

- 100 different items and 1000 shopping carts synthetically generated.
- The number of items per shopping cart: between 0 and 10.
- With increasing minimum support, the runtime decreases as less frequent item sets exist.
- Larger $s_0$, the lower the number of frequent item sets and thus the lower the runtime.
- HyPer operator performs the best,
- implementation in HyPerScript slower than PL/pgSQL (cause: implementation of the array set operator that unnests the array).
Evaluation: PageRank

- The PageRank value was computed for $10^5$ nodes with the same number of edges.
- Implementations in HyPer outperform counterparts in PostgreSQL.
- Scripting language: slightly worse than using a recursive table in PostgreSQL and HyPer.
- Few edges: additional overhead of the integrated operator in HyPer (dictionary for the nodes and edges in a sparse matrix as Compressed-Sparse-Row (CSR)).
- With an increasing number of edges, additional effort amortised, so operator faster than the script function.

![Graph showing runtime comparison]

<table>
<thead>
<tr>
<th>Number of Edges</th>
<th>Runtime in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td>6.80</td>
</tr>
<tr>
<td>$10^2$</td>
<td>4.35</td>
</tr>
<tr>
<td>$10^3$</td>
<td>4.34</td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Threads</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.80</td>
</tr>
<tr>
<td>2</td>
<td>4.35</td>
</tr>
<tr>
<td>4</td>
<td>4.34</td>
</tr>
</tbody>
</table>
Conclusion

- data mining algorithms in SQL-92 by relying on recursive CTEs
- four algorithms: k-Means, DBSCAN, Apriori and PageRank in SQL
- evaluation: worse than the operators in HyPer but similar to MADlib’s library functions
- within recursive tables: support of aggregate and window functions necessary
Thank you for your attention!