Intrusion Detection and Malware Analysis

IDS feature extraction

Pavel Laskov
Wilhelm Schickard Institute for Computer Science
Metric embedding of byte sequences

Sequences
1. blabla blubla blablabu aa
2. bla blablaa bulab bb abla
3. a blabla blabla ablub bla
4. blab blab abba blabla blu

Subsequences

Geometry

Features

Histograms of subsequences
Formalization of embedding

- A sequence $x$ from an alphabet $\Sigma$ of cardinality $N$: $x \in \Sigma^*$
- A language $L$ of pre-defined words:

$$L \subset \Sigma^* = \{w | w \in \Sigma^*\}$$

- Example languages:
  - n-grams
  - “bag-of-words”
  - all subsequences
  - “bag-of-delimiters”

- Embedding function defined over the language:

$$\phi_w(x) : \begin{cases} 
\text{frequency count} \\
\text{binary flag} 
\end{cases} \text{ for } w \text{ in } x$$
Similarity measures for embeddings

Metric embedding enables application of various vectorial similarity measures over sequences, e.g.

<table>
<thead>
<tr>
<th>Kernels</th>
<th>( k(x, y) )</th>
<th>Distances</th>
<th>( d(x, y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>( \sum_{w \in L} \phi_w(x) \phi_w(y) )</td>
<td>Manhattan</td>
<td>( \sum_{w \in L}</td>
</tr>
<tr>
<td>RBF</td>
<td>( \exp(d(x, y)^2 / \sigma) )</td>
<td>Minkowski</td>
<td>( \sqrt[k]{\sum_{w \in L}</td>
</tr>
</tbody>
</table>

### Similarity coefficients

- Jaccard, Kulczynski, . . .
- Chebyshev
  \[ \max_{w \in L} |\phi_w(x) - \phi_w(y)| \]
Abstract similarity measure

- Outer loop:
  \[ s(x, y) = \bigoplus_{w \in L} m(x, y, w) \]

- Inner function:
  \[ m(x, y, w) = \begin{cases} 
  m^+(\phi_w(x), \phi_w(y)), & \text{if } w \text{ matches } x, y \\
  m_x^-(\phi_w(x)), & \text{if } w \text{ mismatches } x \\
  m_y^-(\phi_w(y)), & \text{if } w \text{ mismatches } y 
\end{cases} \]
### Inner function computation

\[ \oplus \quad m^+ (p, q) \quad m^-_x (p) \quad m^-_y (q) \]

<table>
<thead>
<tr>
<th>Kernel functions</th>
<th>$\sum$</th>
<th>$p \cdot q$</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linear</strong></td>
<td>$\sum$</td>
<td>$p \cdot q$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Distances</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manhattan</td>
<td>$\sum$</td>
<td>$</td>
<td>p - q</td>
<td>$</td>
</tr>
<tr>
<td>Minkowski</td>
<td>$\sum$</td>
<td>$(p - q)^k$</td>
<td>$p^k$</td>
<td>$q^k$</td>
</tr>
<tr>
<td>Chebyshev</td>
<td>max</td>
<td>$p - q$</td>
<td>$p$</td>
<td>$q$</td>
</tr>
</tbody>
</table>
How should we store subsequences to ensure linear-time extraction and matching?

- **Hash tables**: simple and relatively efficient; limited embeddings, hash table size difficult to choose.
- **Sorted arrays**: simple and highly efficient (contiguous storage!); limited embeddings
- **Tries**: moderately complex and efficient; limited embeddings.
- **Suffix trees**: unlimited embeddings; very complex, high constants and memory consumption.
- Extract subsequences and store them in an array
- Sort the array
- For any pair of sequences, find matching and mismatching entries by looping over sorted arrays.
- Example: $x = \text{abbaa}$, $y = \text{baaaab}$
How to sort (sub-)sequences?
How to sort (sub-)sequences?

- Radix sort at byte level
  - Simple, linear running time
How to sort (sub-)sequences?

- Radix sort at byte level
  - Simple, linear running time
- Store subsequences in machine words, use numeric sorting
  - Simple, superlinear running time, extremely low constants
How to sort (sub-)sequences?

- Radix sort at byte level
  - Simple, linear running time
- Store subsequences in machine words, use numeric sorting
  - Simple, superlinear running time, extremely low constants
- Ditto, use radix sorting at bit-level
A suffix tree for an $m$-character string $S$ stores all suffixes of $S$.

Properties of suffix trees:

- A suffix tree has exactly $m$ leaves numbered 1 to $m$.
- Each internal node has at least two children.
- Each edge is labeled by a non-empty substring of $S$.
- All edges of the same node begin with different symbols.
- For any leaf $i$, the concatenation of the labels on the path from root to $i$ is the suffix of $S$ starting at position $i$, i.e. $S[i..m]$. 
Suffix tree: a naive storage

$S = ababc$

1 3 2 4

5
What are suffix trees good for?

- **Problem:** Given a string $S$ of length $n$ and a pattern $p$ of length $m$, $m \ll n$, find positions of all occurrences of $P$ in $S$.
- **Classical solution:** $O(m + n)$ (e.g. Knuth-Morris-Pratt)
- **Suffix tree solution:** $O(m)$

![Suffix tree diagram](image)
- Labels are replaced by index ranges.
- Internal nodes contain depth and leaf counts.
- Suffix links point to subtrees corresponding to the next suffix.

$S = \text{ababc}\$
Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$:

$S = ababc$, $P = baaba$

![Suffix Tree Diagram]

- $S = ababc$, $P = baaba$
- The suffix tree shows matching substrings as paths from the root to leaf nodes.
- The substrings are matched as follows:
  - $ab$ at node 1
  - $c$ at node 2
  - $abc$ at node 3
  - $c$ at node 4
Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$:
Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$:

$S = \text{ababc}$, $P = \text{baaaba}$

scan 'a': MATCH, count 1
Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$: 

$S = ababc$, $P = baaaba$

scan 'a': MISMATCH

Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$: 

$S = ababc$, $P = baaaba$

scan 'a': MISMATCH
Chang & Lawler Algorithm

Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$: 

$$S = \text{ababc$}, \ P = \text{baaaba}$$

scan ’a’: MISMATCH
Chang & Lawler Algorithm

Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$:

$S = \text{ababc}$, $P = \text{baaaba}$

scan ’b’: MATCH, count 2
Given a suffix tree for $S$, we can count matching substrings in $S$ and $P$ by walking along $P$ and $S$:

$$S = ababc$, P = baaaba$$

scan ’a’: MATCH, count 1
- A suffix tree for more than one string.
- Creation: concatenate two strings with different delimiters and build a single suffix tree
- Example: GST for \( x = 'abbaa' \) and \( y = 'baaaab' \):

```
  a
 /   \
(2,4) a
   /   /  \
  a b \( a \)$ \( b \)$
  \( a \)$ \( b \)$
 /   /   \   \
(1,3) a b \( a \)$ \( b \)$
   /   /   \   \
(0,2) a b \( a \)$ b$
```

13

**Generalized suffix tree (GST)**
Similarity computation using GST

**function** \( \text{COMPARE}(X, Y : A^*) : IR \)

\[
T \leftarrow \text{CONCAT}(X, Y) \\
S \leftarrow \text{SUFFIXTREE}(T) \\
\text{return } \text{TRAVERSE}(\text{root}[S])
\]

**function** \( \text{TRAVERSE}(v : \text{Node}) : IR \)

\[
s \leftarrow e \\
\text{for } c \leftarrow \text{children}[v] \text{ do} \\
\quad s \leftarrow s \oplus \text{TRAVERSE}(c) \quad \triangleright \text{Depth-first traversal} \\
n \leftarrow \text{FILTER}(\text{length}[v], \text{depth}[v]) \quad \triangleright \text{Filter words on edge to } v \\
s \leftarrow s \oplus m(\phi[v, 1], \phi[v, 2]) \otimes n \\
\text{return } s
\]
Lessons learned

- Extraction of features from packet payloads is tricky but can be efficiently done with specialized data structures.
- In practice, sorted arrays work best for computation of similarity measures.
- Suffix trees are the most powerful data structure for feature extraction: will be used for other problems.
D. Gusfield.  
*Algorithms on strings, trees, and sequences.*  

K. Rieck and P. Laskov.  
Linear-time computation of similarity measures for sequential data.  

E. Ukkonen.  
Online construction of suffix trees.  