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

**Geometry**

**Subsequences**

**Features**

**Histograms of subsequences**

1. a b aa bb blu blubs blub bulab blubla blublabu
2. a b aa bb blu blubs blub bulab blubla blublabu
3. a b aa bb blu blubs blub bulab blubla blublabu
4. a b aa bb blu blubs blub bulab blubla blublabu
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:
  \[ \phi_w(x) : \begin{cases} 
  \text{frequency} \\
  \text{count} \\
  \text{binary flag} 
\end{cases} \text{ for } w \text{ in } x \]
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>
<tr>
<td><strong>Similarity coefficients</strong></td>
<td></td>
<td>Hamming</td>
<td>( \sum_{w \in L} \text{sgn}</td>
</tr>
<tr>
<td>Jaccard, Kulczynski, ...</td>
<td></td>
<td>Chebyshev</td>
<td>( \max_{w \in L}</td>
</tr>
</tbody>
</table>
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>Linear</th>
<th>Distances</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \sum ]</td>
<td>[ p \cdot q ]</td>
<td>0</td>
</tr>
</tbody>
</table>

**Distances**

| Manhattan | \[ \sum \] | \[ |p - q| \] | p | q |
|-----------|-------------|------------------|-----|-----|
| Minkowski | \[ \sum \] | \[ (p - q)^k \] | \[ p^k \] | \[ q^k \] |
| Chebyshev | \[ \max \] | \[ p - q \] | p | q |
How should we store subsequences to ensure linear-time extraction and matching?
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.
Sorted array representation

- 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: $x = \text{abbaa}$, $y = \text{baaaab}$

$\phi[x]$: The length of an array $X$ is denoted by $|X|$. In order to support efficient comparison, the fields of $X$ are sorted by contained words, e.g. using the lexicographical order of the alphabet $A$.

Figure 1 illustrates the sorted arrays of 3-grams extracted from the two example sequences $x$ and $y$.

Algorithm.

Comparison of two sorted arrays $X$ and $Y$ is carried out by looping over the fields of both arrays in the manner of merging sorted arrays (Knuth, 1973). During each iteration the inner function $m$ is computed over contained words and aggregated using the operator $\oplus$. The corresponding comparison procedure in pseudo-code is given in Algorithm 1.

Herein, we denote the case where a word $w$ is present in $x$ and $y$ as match and the case of $w$ being contained in either $x$ or $y$ as mismatch. For run-time improvement, these mismatches can be ignored in Algorithm 1 if a conjunctive similarity measure is computed (cf. Section 3.3).

Algorithm 1

Array-based sequence comparison

1: function $\text{Compare}(X; Y: \text{Array})$
2: $s \leftarrow e$; $i \leftarrow 1$; $j \leftarrow 1$
3: while $i \leq |X|$ or $j \leq |Y|$ do
4:   $x \leftarrow X[i]$; $y \leftarrow Y[j]$
5:   if $y = \text{nil}$ or $\text{word}[x] < \text{word}[y]$ then
6:      $s \leftarrow s \oplus m(\phi[x]; 0)$
7:      $i \leftarrow i + 1$
8:   else if $x = \text{nil}$ or $\text{word}[x] > \text{word}[y]$ then
9:      $s \leftarrow s \oplus m(0; \phi[y])$
10:     $j \leftarrow j + 1$
11:   else
12:      $s \leftarrow s \oplus m(\phi[x]; \phi[y])$
13:      $i \leftarrow i + 1$; $j \leftarrow j + 1$
14: return $s$

Run-time.
The comparison algorithm based on sorted arrays is simple to implement, yet it does not enable linear-time comparison for all embedding languages, e.g. if $L = A^*$. However, sorted arrays enable linear-time similarity measures, if there exist $O(|x|)$ words $w \sqsubseteq x$, which implies all $w \in L$ have no or constant overlap in $x$. Examples are the common bag-of-words and $k$-gram embedding languages.
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 $S$. 

$S = \text{ababc}$
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. $S[i..m]$. 
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**: $O(m)$
- Labels are replaced by index ranges.
- Internal nodes contain depth and leaf counts.
- Suffix links point to subtrees corresponding to the next suffix.

\( S = \text{ababc}\$ \)
Chang & Lawler Algorithm: an example

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

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

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

![Diagram of suffix tree with nodes labeled 1, 2, 3, 4, 5, and edges labeled 'ab', 'b', 'c$'. The root of the tree is labeled 'S = ababc$', and the leaves are labeled '1', '2', '3', '4', '5'.]
Chang & Lawler Algorithm: an example

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'
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
Chang & Lawler Algorithm: an example

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
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: an example

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
Chang & Lawler Algorithm: an example

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

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

scan 'a': MATCH, count 1
Generalized suffix tree (GST)

- A suffix tree for more than one string.
- Creation: concatenate two strings with different delimiters and build a single suffix tree
- Example: GST for \(x = \text{'abbaa'}\) and \(y = \text{'baaaa'}\):
Similarity computation using GST

<table>
<thead>
<tr>
<th>2-grams</th>
<th>“abbaa”</th>
<th>“baaa”</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“abbaa” · “baaa” = 0
Similarity computation using GST

<table>
<thead>
<tr>
<th>2-grams</th>
<th>“abbaa”</th>
<th>“baaaa”</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“abbaa” · “baaaa” = 3
Similarity computation using GST

<table>
<thead>
<tr>
<th>2-grams</th>
<th>“abbaa”</th>
<th>“baaaa”</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“abbaa” · “baaaa” = 3
Similarity computation using GST

<table>
<thead>
<tr>
<th>2-grams</th>
<th>“abbaa”</th>
<th>“baaaa”</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“abbaa” · “baaaa” = 4
Similarity computation using GST

<table>
<thead>
<tr>
<th>2-grams</th>
<th>&quot;abbaa&quot;</th>
<th>&quot;baaaa&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ab</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ba</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bb</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

"abbaa" \cdot "baaaa" = 4
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.  