String matching

Helena Galhardas
DEI IST

String matching – what is it?

- Problem of finding strings that refer to the same real-world entity
  Exs:
  - "David R. Smith" and "David Smith"
  - "1210 W. Dayton St, Madison WI" and "1210 West Dayton, Madison WI 53706"
- Formally:
  - Given two sets of strings $X$ and $Y$, we want to find all pairs of strings $(x, y)$, where $x \in X, y \in Y$ and such that $x$ and $y$ refer to the same real-world entity.
  - These pairs are denoted matches
Challenges (1)

- **Accuracy**
  - Strings referring the same entity are often very different (due to typing/OCR errors, different formatting conventions, abbreviations, nicknames, etc)
  - Solution: define a **similarity measure** $s$ that takes two strings $x$ and $y$ and returns a score in $[0,1]$; $x$ and $y$ match if $s(x,y) \geq t$, being $t$ a pre-specified threshold.

Challenges (2)

- **Scalability**
  - To apply the similarity metric to a large number of strings
  - **Cartesian product** of sets $X$ and $Y$ is quadratic in the size of data – impractical!
  - Solution: to apply the similarity test only to the most promising pairs.
Outline

- Similarity measures
  - Sequence-based
  - Set-based
  - Hybrid
  - Phonetic
- Scaling up string matching

Sequence-based similarity measures

- View the strings as sequences of characters, and compute the cost of transforming one string into the other
  - Edit distance
  - Needleman-Wunch measure
  - Affine Gap measure
  - Smith-Waterman measure
  - Jaro measure
  - Jaro-Winkler measure
Recap. edit distance

- **Levenshtein distance:**
  - Minimum number of operations (insertions, deletions or replacements of characters) needed to transform one string into another

**Ex:** The cost of transforming string “David Smiths” to the string “Davidd Simth” is 4 where the required operations are: deleting a ‘d’, replacing ‘i’ by ‘m’, and deleting an ‘s’.

- Given two strings $s_1$ and $s_2$ and their edit distance, denoted by $d(s_1, s_2)$, the similarity functions is given by

$$s(s_1, s_2) = 1 - \frac{d(s_1, s_2)}{\max(\text{length}(s_1), \text{length}(s_2))}$$

**Ex:** The similarity between “David Smiths” and “David Smiths” is 0.67

Computing the edit distance (recurrence equation)

\[
    d(i,j) = \min\left\{\begin{array}{ll}
    d(i-1,j-1) + c(x_i,y_j) & // \text{copy or substitute} \\
    d(i-1,j) + 1 & // \text{delete } x_i \\
    d(i,j-1) + 1 & // \text{insert } y_j
    \end{array}\right.
\]

$c(x_i,y_j) = 0$ if $x_i = y_j$, 1 otherwise

$d(0,0) = 0; \ d(i,0) = i; \ d(0,j) = j$
Dynamic programming matrix

Computing the distance between “dva” and “dave”

Cost of computing $d(x,y)$ is $\text{length}(x) \times \text{length}(y)$.

Needleman-Wunch measure

- Models the variations between strings more explicitly than a series of transformations
  - Allows different penalties for variations between strings (e.g., transforming ‘o’ into 0 receives a smaller penalty than transforming ‘a’ to 0)
  - Explicitly models gaps in the alignment of the two strings, and assigns an arbitrary cost to the gap
Computing the Needleman-Wunch measure

- Assigns a **score** to each alignment between the two input strings and chooses the score of the best alignment
  - Alignment between two strings \( x \) and \( y \): set of correspondences between the characters of \( x \) and \( y \), allowing for gaps
  - Ex: one possible alignment between \( x = \text{dva} \) and \( y = \text{deeve} \)

\[
\begin{array}{c}
d - - v a \\
d e e v e
\end{array}
\]

Computing the score of the alignment (1)

- **Using a score matrix**
  - Assigns a score for a correspondence between every pair of characters; penalizes transformations on a case by case basis.
  - Ex: A **correspondence of two identical characters** may score 2; or -1 otherwise
- **and a gap penalty**
  - A **gap** of length 1 has a penalty \( cg \) (ex: 1); a gap of length \( k \) has a penalty \( k \times cg \)
Example (1)

- Given the score matrix:

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>v</th>
<th>a</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>v</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>a</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
</tr>
</tbody>
</table>

- And the gap penalty:

\[ c_g = 1 \]

Computing the score of the alignment (2)

- Given an alignment A between strings x and y, a score matrix \( c(x_j, y_j) \), and a gap penalty \( c_g \), the score of \((x,y)\) given A is the sum of the scores of all matches in A minus the penalties for the gaps.

Ex: score of the alignment:

\[
\begin{align*}
2 \text{ (for match d-d)} &+ 2 \text{ (for match v-v)} - 1 \text{ (for match a-e)} - 2 \text{ (penalty for the gap of length 2)} = 1
\end{align*}
\]
Computing the Needleman-Wunch score (recurrence equation)

\[
s(i,j) = \max \begin{cases} 
  s(i-1,j-1) + c(x_i, y_j) \\
  s(i-1,j) - c_g \\
  s(i,j-1) - c_g \\
\end{cases}
\]

Initialization:

\[
s(0,j) = -j c_g \\
 s(i,0) = -i c_g
\]

Example (2)

- Dynamic programming matrix between “dva” and “deeve”:

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>e</th>
<th>e</th>
<th>v</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>d</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>v</td>
<td>-2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>-3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Optimal alignment: \( d \ - \ - v \ a \)

\[
\begin{array}{cccccc}
  & d & e & e & v & e \\
 0 & 0 & -1 & -2 & -3 & -4 & -5 \\
d & -1 & 2 & -1 & 0 & -1 & -2 \\
v & -2 & 1 & 1 & 0 & 2 & 1 \\
a & -3 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
\]
Jaro measure

- Developed mainly to compare short strings, such as first and last names
- Given two strings $x$ and $y$,
  - Find the common chars $x_i$ and $y_i$ such that $x_i = y_j$ and $|i - j| \leq \min\{|x|, |y|\}/2$
    - Common chars: Those that are identical and are positionally "close to one another"
    - The number of common chars $x_i$ in $x$ and $y_i$ in $y$ is the same – it is called $c$
  - Compare the $i$-th common character of $x$ with the $i$-th common character of $y$. If they don’t match, then there is a transposition. Number of transpositions is $t$
  - Compute the Jaro score as:
    $$jaro(x,y) = \frac{1}{3} \left[ \frac{c}{|x|} + \frac{c}{|y|} + \frac{(c - t/2)}{c} \right]$$

Cost of computation: $O(|x||y|)$

Example

- $x = \text{jon}; \ y = \text{john}$
  - $c = 3$
  - Common character sequence in $x$ and $y$ is: jon
  - Nb. transpositions, $t = 0$
  - Jaro$(x,y) = \frac{1}{3} (3/3 + \frac{3}{4} + 3/3) = 0.917$
  - Edit distance$(x,y) = 0.75$
- $x = \text{jon}; \ y = \text{ojn}$
  - $C = 3$
  - Common character sequence in $x$: jon
  - Common character sequence in $y$: ojn
  - $t = 2$
  - Jaro$(x,y) = \frac{1}{3} (3/3 + \frac{3}{4} + (3-2/2)/3) = 0.81$
Jaro-Winkler measure

- Modifies the Jaro measure by adding more weight to a common prefix
- Introduces two parameters:
  - \( PL \): length of the longest common prefix between the two strings
  - \( PW \): weight to give the prefix

\[
\text{Jaro-Winkler}(x,y) = (1 – PL*PW) * \text{jaro}(x,y) + PL*PW
\]

Set-based similarity measures

- View the strings as sets or multi-sets of tokens, and use set-related properties to compute similarity scores
  - Overlap measure
  - Jaccard measure
  - TF-IDF measure
- Several ways of generating tokens from strings
  - Words in the string (delimited by space char)
    - Tokens of “david smith”: {david, smith}
  - Q-grams: substrings of length \( q \) that are present in the string
    - 3-grams of “david”: {#da, dav, avi, vid, id#}
Overlap measure

- Let $B_x$ and $B_y$ be the sets of tokens generated for strings $x$ and $y$
  - Overlap measure: returns the number of common tokens
    \[ O(x, y) = |B_x \cap B_y| \]
  - Ex: $x = \text{dave}$; $y = \text{dav}$
    - Set of all 2-grams of $x$: $B_x = \{\#d, \text{da}, \text{av}, \text{ve}, \text{e}\}$
    - Set of all 2-grams of $y$: $B_y = \{\#d, \text{da}, \text{av}, \text{v}\}$
    - $O(x, y) = 3$

Jaccard measure

- Jaccard similarity score between two strings $x$ and $y$ is:
  \[ J(x, y) = \frac{|B_x \cap B_y|}{|B_x \cup B_y|} \]
  - Ex: $x = \text{“dave”}$ with $B_x = \{\#d, \text{da}, \text{av}, \text{ve}, \text{e}\}$
    - $y = \text{“dav”}$ with $B_y = \{\#d, \text{da}, \text{av}, \text{v}\}$
    - $J(x, y) = \frac{3}{6}$
TF/IDF measure

**Intuition:** two strings are similar if they contain common distinguishing terms

Ex: \( x = \) “Apple Corporation, CA”
    \( y = \) “IBM Corporation, CA”
    \( z = \) “Apple Corp.”

- Edit distance and Jaccard measure would match \( x \) and \( y \)
- TF/IDF is able to recognize that “Apple” is a distinguishing term, whereas “Corporation” and “CA” are not

**Definitions**

- Each string is converted into a bag of terms (a document in IR terminology)
  Ex: \( x = \) aab; \( y = \) ac; \( z = \) a
  string \( x \) is converted into document \( B_x = \{ a, a, b \} \)

- For every term \( t \) and document \( d \), compute:
  - **Term frequency, \( tf(t, d) \):** number of times \( t \) occurs in \( d \)
    - \( tf(a, x) = 2 \)
  - **Inverse document frequency, \( idf(t) \):** total number of documents in the collection divided by the number of documents that contain \( t \)
    - \( idf(a) = 3/3 \)
More definitions

- Each document \( d \) is represented into a feature vector \( v_d \)
  - Vector \( v_d \) has a feature \( v_d(t) \) for each term \( t \), and the value of \( v_d(t) \) is a function of the TF and IDF scores
  - Vector \( v_d \) has as many features as the number of terms in the collection
- Two documents are similar if their corresponding vectors are close to each other

Example

\( x = \text{aab}; y = \text{ac}; z = \text{a} \)
\( B_x = \{\text{a, b}\}; B_y = \{\text{a, c}\}; B_z = \{\text{a}\} \)
\( \text{tf(a, x)} = 2; \text{tf(b, x)} = 1; \ldots \text{tf(c, z)} = 0 \)
\( \text{idf(a)} = 3/3 = 1; \text{idf(b)} = 3/1 = 3; \text{idf(c)} = 3/1 \)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_x )</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( v_y )</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>( v_z )</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

with \( v_d(t) = \text{tf(t, d)} \cdot \text{idf(t)} \)
Computing the TF/IDF similarity score

- Given two strings $p$ and $q$
- Let $T$ be the set of all terms in the collection
- Vectors $v_p$ and $v_q$ can be viewed as vectors in the $|T|$-dimensional space, where each dimension corresponds to a term

**TF/IDF score** between $p$ and $q$ is the cosine of the angle between these two vectors:

$$S(p, q) = \frac{\sum_{t \in T} v_p(t) \cdot v_q(t)}{\sqrt{\left(\sum_{t \in T} v_p(t)^2\right) \cdot \left(\sum_{t \in T} v_q(t)^2\right)}}$$

Ex: $s(x, y) = \frac{2 \cdot 3}{\sqrt{2^2 + 3^2} \cdot \sqrt{3^2 + 3^2}}$

Observations

- TF/IDF similarity score between two strings is **high** if they share many frequent terms (terms with high TF scores), unless these terms also commonly appear in other strings in the collection (terms with low IDF scores)
- An alternative score computation for dampening the TF and IDF components by a log factor is:

$$v_d(t) = \log(\text{tf}(t, d) + 1).\log(\text{idf}(t))$$

- With $v_d(t)$ normalized to length 1:
  - $v_d(t) = v_d(t) / \sqrt{\sum_{t \in T} v_d(t)^2}$
Hybrid similarity measures

- Combine the benefits of sequence-based and set-based methods
  - Generalized Jaccard measure
    - Enables approximate matching between tokens
  - Soft TF/IDF similarity measure
    - Similar to Generalized Jaccard measure, but using TF/IDF
  - Monge-Elkan similarity measure
    - Breaks both strings into sub-strings and then applies a similarity function to each pair of sub-strings

Phonetic similarity measures

- Match strings based on their sound
  - Specially effective in matching names (e.g., “Meyer” and “Meier”), which are often spelled in different ways but sound the same
  - Most commonly used similarity measure: soundex
    - Maps a surname \( x \) into a four-character code that captures the sound of the name
    - Two surnames are considered similar if they share the same code
Mapping a surname into a code (1)

Ex: \( x = \text{Ashcraft} \)

1. Keep the first letter of \( x \) as the first letter of the code
   - Ex: First letter of \( x \) is A

2. Remove all occurrences of W and H. Go over the remaining letters and replace them with digits as follows:
   - Replace B, F, P, V with 1
   - Replace C, G, J, K, Q, S, X, Z with 2
   - Replace D, T with 3
   - Replace L with 4
   - Replace M, N with 5
   - Replace R with 6

Mapping a surname into a code (2)

- Do not replace the vowels A, E, I, O, U
- Ex: Ashcraft -> A226a13

3. Replace each sequence of identical digits by the digit itself
   - A226a13 -> A26a13

4. Drop all the non-digit letters, except the first one. Return the first four letters as the soundex code
   - A26a13 -> A261
Observations about the soundex code

- Is always a letter followed by three digits, padded by 0 if there are not enough digits
  Ex: soundex of Sue is S000
- Hashes similar sounding consonants (such as B, F, P, and V) into the same digit, which means it maps similar sounding names into the same soundex code
  Ex: Both Robert and Rupert map into R163
- Is not perfect
  Ex: fails to map Gough and Goff into the same code
- Widely used to match names in census records, vital records, genealogy databases
  - Works well for names from different origins
  - Doesn’t work well for Asian names, because the discriminating power of these names is based on vowels that are ignored by the code

A better phonetic similarity measure

- Metaphone
  - A string is converted into a code with variable size
  - It takes into account English pronouncing rules
- Improved versions of the algorithm:
  - Double metaphone
  - Metaphone 3
Outline

✓ Similarity measures
  ✓ Sequence-based
  ✓ Set-based
  ✓ Hybrid
  ✓ Phonetic

➢ Scaling up string matching
  ➢ Inverted index over strings
  ➢ Size filtering
  ➢ Prefix filtering
    • Position filtering
    • Bound filtering

Recap. Challenge (2)

■ Scalability
  ❑ To apply the similarity metric to a large number of strings
  ❑ Cartesian product of sets X and Y is quadratic in the size of data – impractical!
  ❑ Solution: to apply the similarity test only to the most promising pairs.
Naïve matching solution

for each string $x \in X$
    for each string $y \in Y$
        if $s(x,y) \geq t$, return $(x,y)$ as a matched pair

Computational cost: $O(|X||Y|)$ is impractical for large data sets.

Solution: Blocking

- To develop a method $\text{FindCands}$ to quickly find the string that may match a given string $x$
- Then, use the following algorithm:
  for each string $x \in X$
      use a method $\text{FindCands}$ to find a candidate set $Z \subseteq Y$
      for each string $y \in Z$
          if $s(x,y) \geq t$, return $(x,y)$ as a matched pair
  - Takes $O(|X||Z|)$ time, much faster than $O(|X||Y|)$, because $|Z|$ is much smaller than $|Y|
  - Set $Z$ should contain all true positives and as few negative positives as possible
Techniques used in FindCands

- Inverted indexes over strings
- Size filtering
- Prefix filtering
- Position filtering

- Explained using the Jaccard and Overlap measures

Inverted index over strings

1. Converts each string $y \in Y$ into a document $D(y)$, then builds an inverted index $I_y$ over these documents
2. Given a term $t$, use $I_y$ to quickly find the list of documents created from $Y$ that contain $t$, which gives the strings $y \in Y$ that contain $t$
Example

Two sets of strings X and Y to be matched:

Set X
1: \{lake, mendota\}
2: \{lake, monona, area\}
3: \{lake, mendota, monona, dane\}

Set Y
4: \{lake, monona, university\}
5: \{monona, research, area\}
6: \{lake, mendota, monona, area\}

Inverted index \(I_Y\)

<table>
<thead>
<tr>
<th>Terms in Y</th>
<th>ID Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>5, 6</td>
</tr>
<tr>
<td>lake</td>
<td>4, 6</td>
</tr>
<tr>
<td>mendota</td>
<td>6</td>
</tr>
<tr>
<td>monona</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>research</td>
<td>5</td>
</tr>
<tr>
<td>university</td>
<td>4</td>
</tr>
</tbody>
</table>

Given a string \(x = \{lake, mendota\}\) \textbf{FindCands} uses \(I_Y\) to find and merge the ID lists for \textit{lake} and \textit{mendota} and obtain \(Z = \{4, 6\}\)

Limitations

- Inverted list of some terms (e.g., stopwords) can be very long
  - Building and manipulating such lists are quite costly
- Requires enumerating all pairs of strings that share at least one term
  - The set of such pairs can still be very large
Size filtering

- Retrieves only the strings in Y whose size makes them match candidates
  - Given a string \( x \in X \), infer a constraint on the size of strings in Y that can possibly match \( x \)
  - The filter uses a B-tree index to retrieve only the strings that fit the size constraints

Derivation of constraints on the size of strings

\[
J(x,y) = \frac{|x \cap y|}{|x \cup y|}
\]

\[
\frac{1}{J(x,y)} \geq \frac{|y|}{|x|} \geq J(x,y) \quad \text{(can be proved)}
\]

If \( x \) and \( y \) match, then

\[
J(x,y) \geq t
\]

So:

\[
\frac{1}{t} \geq \frac{|y|}{|x|} \geq t \iff \\
\frac{|x|}{t} \geq |y| \geq |x|.t
\]

Given a string \( x \in X \), only the strings that satisfy this Equation can possible match \( x \)
Example

\[ x = \{\text{lake, mendota}\} \]
\[ t = 0.8 \]
Using the equation: \( |x|/t \geq |y| \geq |x| \cdot t \)
If \( y \in Y \) matches \( x \), we must have:
\[ 2/0.8 = 2.5 \geq |y| \geq 2 \cdot 0.8 = 1.6 \]
So, none of the strings in the set \( Y \) satisfies this constraint!

Set \( Y \)
4: \{lake, monona, university\}
5: \{monona, research, area\}
6: \{lake, mendota, monona, area\}

B-tree index

- Procedure \texttt{FindCands} builds a B-tree over the sizes of strings in \( Y \)
- Given a string \( x \in X \), it uses the index to find strings in \( Y \) that satisfy equation:
  \[ |x|/t \geq |y| \geq |x| \cdot t \]
- Returns that set of strings that fit the size constraint
  - Effective when there is significant variability in the number of tokens in the strings \( X \) and \( Y \)
Prefix filtering

**Idea:** If two sets share many terms, then large subsets of them also must share terms

- This way, the number of candidate strings that may match a string \( x \) can be reduced

**Using the overlap measure, if**

- \( |x \cap y| \geq k \), then
- Any subset \( x' \subseteq x \) of size at least \( |x|-(k-1) \) must overlap \( y \)

---

**Example**

\( x = \{ \text{lake}, \text{monona}, \text{area} \} \)
\( y = \{ \text{lake}, \text{mendota}, \text{monona}, \text{area} \} \)

\( |x \cap y| \geq 2 \)

Thus, the subset:

\( x' = \{ \text{lake}, \text{monona} \} \) overlaps \( y \)

as does any other subset of size 2 of \( x \)
**Procedure FindCands**

- **Goal:** to find all pairs \((x, y)\) with overlap \(O(x, y) \geq k\)
- **Given a particular set** \(x\), construct a subset \(x'\) of size \(|x|-(k-1)\) and use an inverted index to find all sets \(y\) that overlap \(x'\)

<table>
<thead>
<tr>
<th>Set X</th>
<th>Terms in Y</th>
<th>ID Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: {lake, mendota}</td>
<td>area</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>2: {lake, monona, area}</td>
<td>lake</td>
<td>4, 6</td>
</tr>
<tr>
<td>3: {lake, mendota, monona, dane}</td>
<td>mendota</td>
<td>6, 7</td>
</tr>
<tr>
<td></td>
<td>monona</td>
<td>4, 5, 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set Y</th>
<th>Terms in Y</th>
<th>ID Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>4: {lake, monona, university}</td>
<td>research</td>
<td>5</td>
</tr>
<tr>
<td>5: {monona, research, area}</td>
<td>university</td>
<td>4</td>
</tr>
<tr>
<td>6: {lake, mendota, monona, area}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: {dane, area, mendota}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Terms in Y</th>
<th>ID Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>lake</td>
<td>4, 6</td>
</tr>
<tr>
<td>mendota</td>
<td>6, 7</td>
</tr>
<tr>
<td>monona</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>research</td>
<td>5</td>
</tr>
<tr>
<td>university</td>
<td>4</td>
</tr>
</tbody>
</table>

**Example**

Suppose we want to match strings in \(X\) and \(Y\) using \(O(x, y) \geq 2\)

**Inverted index** \(I_y\)

<table>
<thead>
<tr>
<th>Terms in Y</th>
<th>ID Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>5, 6, 7</td>
</tr>
<tr>
<td>lake</td>
<td>4, 6</td>
</tr>
<tr>
<td>mendota</td>
<td>6, 7</td>
</tr>
<tr>
<td>monona</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>research</td>
<td>5</td>
</tr>
<tr>
<td>university</td>
<td>4</td>
</tr>
</tbody>
</table>

Given the string \(x_1 = \{lake, mendota\}\), take the prefix with size \(|x_1| - 1\), which is \{lake\} = \(x_1'\).

Then, use the inverted index \(I_y\) to find all the strings in \(Y\) that contain at least a token in \(x_1'\), which gives \{y_4, y_6\}
References