Bachelor Thesis: Fast Approximate Title Matching

Mirko Brodesser

University of Freiburg, Department of Computer Science
Chair of Algorithms and Data Structures
Prof. Dr. Hannah Bast, Marjan Celikik

21. September 2010

Problem Definition

- Given a large set of clean records (titles) and a query
- We want the title with the largest similarity to the query; in the shortest possible time
- We look at some examples...

Problem Definition

- Given a large set of clean records (titles) and a query
- We want the title with the largest similarity to the query; in the shortest possible time
- We look at some examples...

The different types of errors:

- Missing words
- Additional words
- Spelling mistakes
- Concatenations

Similarity measures: Jaccard similarity

We need a similarity measure which takes these types of errors into account.

Similarity measures: Jaccard similarity

We need a similarity measure which takes these types of errors into account.

One possibility: **Jaccard similarity:**
Q, R are sets of tokens ($=$ either words or n-grams)
$J(Q, R) = \frac{|Q \cap R|}{|Q \cup R|}$

Similarity measures: Jaccard similarity

We need a similarity measure which takes these types of errors into account.

One possibility: **Jaccard similarity:**
Q, R are sets of tokens ($=$ either words or n-grams)
$J(Q,R) = \frac{|Q \cap R|}{|Q \cup R|}$

Example:

Query:	almostfamous trash	Matching 3-grams
Record1:	almost famous	{alm, lmo, mos, ost, fam, amo, mou, ous}
Record2:	the trash story	{tra, ras, ash}

Similarity measures: Jaccard similarity

We need a similarity measure which takes these types of errors into account.

One possibility: **Jaccard similarity:**
Q, R are sets of tokens (= either words or n-grams)
$J(Q,R) = \frac{|Q \cap R|}{|Q \cup R|}$

Example:

Query:	almostfamous trash	Matching 3-grams
Record1:	almost famous	{alm, lmo, mos, ost, fam, amo, mou, ous}
Record2:	the trash story	{tra, ras, ash}

$\Rightarrow J(Query, Record1) = \frac{8}{13} > J(Query, Record2) = \frac{3}{17}$

Jaccard similarity

Assume the query has two spellings mistakes:
Example:

Query:	almustfamuus trash	Matching 3-grams

Jaccard similarity

Assume the query has two spellings mistakes:
Example:

Query:	almustfamuus trash	Matching 3-grams
Record1:	almost famous	{alm, ~~lmo~~, ~~mos~~, ~~ost~~, fam, ~~amo~~, ~~mou~~, ~~ous~~}
Record2:	the trash story	{tra, ras, ash}

Jaccard similarity

Assume the query has two spellings mistakes:
Example:

Query:	almustfamuus trash	Matching 3-grams
Record1:	almost famous	{alm, ~~lmo~~, ~~mos~~, ~~ost~~, fam, ~~amo~~, ~~mou~~, ~~ous~~}
Record2:	the trash story	{tra, ras, ash}

$\Rightarrow J(Query, Record1) = \frac{2}{13} < J(Query, Record2) = \frac{3}{17}$

Observe: *Position* of a spelling mistake influences similarity

Weighted Jaccard similarity

3) Our similarity measure, **weighted Jaccard similarity:**

R = Set of words from the record.
Q = Set of non-overlapping substrings from the 'normalized' query with the record, for example:

Weighted Jaccard similarity

3) Our similarity measure, **weighted Jaccard similarity:**

R = Set of words from the record.
Q = Set of non-overlapping substrings from the 'normalized' query with the record, for example:

Query:	almostfamous trash
Record:	almost famous
Q:	$\{$almost, famous$\}$

Weighted Jaccard similarity

3) Our similarity measure, **weighted Jaccard similarity:**

R = Set of words from the record.
Q = Set of non-overlapping substrings from the 'normalized' query with the record, for example:

Query:	almostfamous trash
Record:	almost famous
Q:	{almost, famous}

$WJ(Q,R) = \frac{W(Q \cap R)}{W(Q \cup R)}$, where $W(S) = \sum_{s \in S} w(s)$
and $w(s)$ = s.length() - punishment(s).

An existing algorithm: ppjoin

Xiao et al. described an algorithm called **ppjoin:**

- allows to use 3-grams
- allows to use Jaccard similarity
- has a threshold parameter for the similarity

An existing algorithm: ppjoin

Xiao et al. described an algorithm called **ppjoin:**

- allows to use 3-grams
- allows to use Jaccard similarity
- has a threshold parameter for the similarity

Basic idea:

- Pre-process records: build inverted index over the 3-grams of the words
- Index depends on the threshold. Example: threshold $= 1.0 \Rightarrow$ Only one 3-gram per record has to be indexed
- Create the candidate set from the 3-grams of the query
- Apply different filters, for example *size filtering* to reduce the candidate set.

An existing algorithm: ppjoin

Xiao et al. described an algorithm called **ppjoin:**

- allows to use 3-grams
- allows to use Jaccard similarity
- has a threshold parameter for the similarity

Basic idea:

- Pre-process records: build inverted index over the 3-grams of the words
- Index depends on the threshold. Example: threshold $= 1.0 \Rightarrow$ Only one 3-gram per record has to be indexed
- Create the candidate set from the 3-grams of the query
- Apply different filters, for example *size filtering* to reduce the candidate set.

Problem in our case: low threshold required $\rightsquigarrow$ large inverted listes $\rightsquigarrow$ long running times.

Our algorithm: atMatch

Our algorithm: **atMatch** (**a**pproximate **t**itle **M**atch)

- Uses our Weighted Jaccard similarity

Basic idea:

- Pre-processing: index all *words* from the records, except *stopwords*

Our algorithm: atMatch

Our algorithm: **atMatch** (**a**pproximate **t**itle **M**atch)

- Uses our Weighted Jaccard similarity

Basic idea:

- Pre-processing: index all *words* from the records, except *stopwords*
- Query: find all valid approximate substrings. Example:

Our algorithm: atMatch

Our algorithm: **atMatch** (**a**pproximate **t**itle **M**atch)

- Uses our Weighted Jaccard similarity

Basic idea:

- Pre-processing: index all *words* from the records, except *stopwords*
- Query: find all valid approximate substrings. Example:

Original query:	t	h	e	f	a	s	t	f	i	r	i	o	u	s
Approx. substr.:								f	u	r	i	o	u	s
Approx. substr.:	t	h	e	f	t									
Approx. substr.:			f	a	u	s	t							
Approx. substr.:				f	a	s	t							
Approx. substr.:	t	h	e											
Approx. substr.:					a									

Our algorithm: atMatch

- Candidate set is generated from the inverted lists of the valid approximate substring, in our example:

Candidate Id	Record
C1	*the fast and the furious*
C2	*i am furious*
C3	*go fast*

Our algorithm: atMatch

- Candidate set is generated from the inverted lists of the valid approximate substring, in our example:

Candidate Id	Record
C1	*the fast and the furious*
C2	*i am furious*
C3	*go fast*

- Then match all valid approximate substrings with the candidates:

Candidate Id	Record
C1	**the fast** and the **furious**
C2	i am **furious**
C3	go **fast**

Our algorithm: atMatch

- Candidate set is generated from the inverted lists of the valid approximate substring, in our example:

Candidate Id	Record
C1	*the fast and the furious*
C2	*i am furious*
C3	*go fast*

- Then match all valid approximate substrings with the candidates:

Candidate Id	Record
C1	**the fast** and the **furious**
C2	i am **furious**
C3	go **fast**

- Calculate the record with highest weighted Jaccard similarity from the candidates

Our algorithm: atMatch

The valid approximate substrings are tried to match by *decreasing length*

- Reason: We always want to allow the largest substrings to match, for example:

Query:	casablanca
Valid approx. substrings:	casablanca, casa
Candidate record:	la casa vianello casablanca

Our algorithm: atMatch

The valid approximate substrings are tried to match by *decreasing length*

- Reason: We always want to allow the largest substrings to match, for example:

Query:	casablanca
Valid approx. substrings:	casablanca, casa
Candidate record:	la casa vianello casablanca

- Disadvantage: greedy $\rightsquigarrow$ not optimal. Example:

Query:	abcdef
Candidate record:	abc abcde def

Our algorithm: atMatch

The valid approximate substrings are tried to match by *decreasing length*

- Reason: We always want to allow the largest substrings to match, for example:

Query:	casablanca
Valid approx. substrings:	casablanca, casa
Candidate record:	la casa vianello casablanca

- Disadvantage: greedy $\rightsquigarrow$ not optimal. Example:

Query:	abcdef
Candidate record:	abc abcde def

But: this case is expected to be rare

Experimental results

We compare **ppjoin** (using Jaccard similarity) with our algorithm **atMatch**:

1) IMDB titles, about 1.5 million records, 109 queries (filenames):

Algorithm	**Average elapsed time**	**Correct assignments**
PPJOIN-0.05	468.69 ms	69.72 %
PPJOIN-0.1	201.88 ms	69.72 %
PPJOIN-0.2	107.34 ms	64.22 %
PPJOIN-0.3	60.89 ms	54.13 %
PPJOIN-0.4	39.29 ms	44.59 %
PPJOIN-0.5	22.23 ms	24.77 %
PPJOIN-0.6	10.53 ms	8.25 %
PPJOIN-0.7	4.38 ms	2.75 %
ATMATCH	46.20 ms	78.90 %

Experimental results: DBLP

2) DBLP titles, about 1.5 million records, 100 randomly chosen queries with different added types of errors:

1. **Typos:** For each query, we randomly changed one letter/number per 10 characters.
2. **Adding words:** For each query, we randomly added one word per 12 characters.
3. **Removing words:** For each query, we randomly removed one word per 15 characters.
4. **Concatenations:** For each query, we randomly added a concatenation of two words per 6 characters.

Experimental results: DBLP

TE = Typos, AE = Added words, RE = Rem. words, CE = Concat.

Algorithm	**Avg. time**	**Corr. assignm.**	**TE**	**AE**	**RE**	**CE**
PPJOIN-0.1	3541.73 ms	99 %	T	T	F	F
PPJOIN-0.2	1758.90 ms	99 %	T	T	F	F
PPJOIN-0.3	1103.48 ms	97 %	T	T	F	F
ATMATCH	1205.17 ms	100 %	T	T	F	F
PPJOIN-0.1	1857.43 ms	48 %	T	T	T	F
PPJOIN-0.2	968.91 ms	48 %	T	T	T	F
PPJOIN-0.3	553.75 ms	36 %	T	T	T	F
ATMATCH	278.51 ms	52 %	T	T	T	F
PPJOIN-0.1	1533.88 ms	44 %	T	T	T	T
PPJOIN-0.2	782.76 ms	44 %	T	T	T	T
PPJOIN-0.3	594.46 ms	34 %	T	T	T	T
ATMATCH	272.63 ms	52 %	T	T	T	T

Possible improvements

1) Considering the ordering of the words, for example:

Id	Record
R1	date movie 2006
R2	movie date 2006

Possible improvements

1) Considering the ordering of the words, for example:

Id	Record
R1	date movie 2006
R2	movie date 2006

2) Popularity: For example if two movies have the same similarity, choose the more popular one

Query:	aspirin flyboys	Popularity
Candidate 1:	aspirin 2006	5 votes
Candidate 2:	flyboys 2006	13938 votes

Possible improvements

3) Ignore certain valid approximate substrings, for example:

Original query:	h	a	n	g	o	v	e	r
Valid correct substr.:	h	a	n	g	o	v	e	r
Valid correct substr.:	h	a	n	g				
Valid correct substr.:					o	v	e	r
Valid approx. substr.:				c	o	v	e	r

Possible improvements

3) Ignore certain valid approximate substrings, for example:

Original query:	h	a	n	g	o	v	e	r
Valid correct substr.:	h	a	n	g	o	v	e	r
Valid correct substr.:	h	a	n	g				
Valid correct substr.:					o	v	e	r
Valid approx. substr.:				c	o	v	e	r

Idea: ignore approximate substrings of "long" (e.g. length ≥ 8) correct substrings.

Thank you for your attention!