

Analiticcl

Maarten van Gompel

2022-01-20

Introduction

What is analiticcl?

- ▶ Analiticcl is a string-matching / fuzzy-matching system
- ▶ Intended for text normalisation like:
 - ▶ diachronic spelling variation
 - ▶ post-OCR/HTR variation
 - ▶ spelling correction (especially non-word errors)
- ▶ Lexicon-based; fuzzy lookups against a lexicon, using a smart *heuristic*

Introduction: Context

- ▶ Developed in the Golden Agents projects
- ▶ Builds upon prior research (Reynaert 2010; Reynaert 2004)

Introduction: Implementation

- ▶ Built with performance and scalability in mind
 - ▶ Multi-threaded (parallelisation)
 - ▶ Written in Rust, compiles to native code
 - ▶ Low-level command-line tool and programming library
 - ▶ for Rust and for Python, via a binding
 - ▶ Unit/integration tests, CI, benchmarks
- ▶ Feature-rich
 - ▶ Highly parametrised
 - ▶ Flexible usage
- ▶ Source: <https://github.com/proycon/analiticcl>
- ▶ License: GNU GPLv3

Installation

Download, compile and install:

```
$ cargo install analiticcl
```

and/or for the Python binding:

```
$ pip install analiticcl
```

Fuzzy string matching

Core function: Fuzzy string matching

- ▶ Given a string, find the forms in the lexicon that are closest (**query** mode)
 - ▶ *error correction*
- ▶ Given a text, find corrections for arbitrary substrings in the text (**search** mode)
 - ▶ *error detection and correction*
- ▶ Given a lexicon entry, find close variants in the text (**learn** mode)

Example: Query mode

```
$ analiticcl query --interactive --lexicon examples/eng.aspell.lexicon  
                  --alphabet examples/simple.alphabet.tsv
```

```
Initializing model...
```

```
Loading lexicons...
```

```
Building model...
```

```
Computing anagram values for all items in the lexicon...
```

```
- Found 119773 instances
```

```
Adding all instances to the index...
```

```
- Found 108802 anagrams
```

```
Creating sorted secondary index...
```

```
Sorting secondary index...
```

```
...
```

```
Querying the model...
```

```
(accepting standard input; enter input to match, one per line)
```

```
seperate
```

seperate	separate	0.734375	\	
	operate	0.6875	\	
	desperate	0.6875	\	
	temperate	0.6875		serrate 0.65625

Variant matching: Naive approach

A naive approach to variant matching:

- ▶ Given m input words
- ▶ Compute edit distance (levenshtein) between each input word to all words in the lexicon (n)
- ▶ High computational cost! $O(mn)$, and the levenshtein algorithm itself has already a $O(l)$ (l =length) time complexity.
 - ▶ Does not scale

Variant matching: anagram hashing (1)

Anagram hashing (Reynaert 2010; Reynaert 2004) aims to drastically reduce the variant search space.

- ▶ Provides a fast *heuristic* for edit distance
- ▶ Analiticcl reimplements and improves upon the idea implemented in earlier tool TICCL

Variant matching: anagram hashing (2)

An Anagram Value (AV)..

- ▶ is computed for each 'word' in the input and in the lexicon
- ▶ uniquely represents the combination of characters in the word (unordered)
 - ▶ $AV(east) = AV(seat) = AV(eats)$
- ▶ has compositional properties:
 - ▶ $AV(eat) \cdot AV(s) = AV(eats)$
 - ▶ $\frac{AV(eats)}{AV(s)} = AV(eat)$
- ▶ each anagram value can be unambiguously decomposed to all its constituents
- ▶ no collisions between anagrams guaranteed (in this reimplementation)
- ▶ anagrams themselves deliberately collide
- ▶ serves as the key in a hash map (stores the lexicon)

Variant matching: hash function

Computation of the Anagram Value is simple composition of **prime** factors:

- ▶ **Input:** alphabet
- ▶ Each 'letter' in the alphabet is assigned a successive **prime number**, this is the Anagram Value of the 'letter'.
 - ▶ Example:
 $AV(a) = 2, AV(b) = 3, AV(c) = 7, AV(d) = 11, AV(e) = 17$
 - ▶ The use of prime number guarantees no collisions between anagrams
 - ▶ Novel with respect to Reynaert's approach.
- ▶ Simple hashing function (l =length, c_i =character at index i):

$$\prod_{i=0}^l AV(c_i)$$

- ▶ **Caveat:** May result in very large integers!
 - ▶ Exceeds 64-bit register
 - ▶ Requires an efficient big integer implementation

Variant matching: Search (1)

Loading stage: Compute Anagram Index and secondary index

- ▶ Compute Anagram Value for each entry in the lexicon and store in a hash map (the anagram index)
- ▶ Mapping the anagram value to all instances of the anagram:

$$AV(a, e, s, t) \mapsto [east, seat, eats]$$

- ▶ Compute a secondary index mapping to *sorted anagram values*:

$$(n, |s|) \mapsto L$$

- ▶ where s is a string, $|s|$ its length in characters, and n it's length in words/tokens
- ▶ where L is a *sorted* list of anagram values
- ▶ example: $(1, 4) \mapsto [AV(a, e, s, t), \dots]$

Variant matching: Search (2)

Search stage: Given a 'word' to correct:

- ▶ we compute the anagram value for the input
- ▶ we look up this anagram value in the anagram index (if it exists) and gather the variant candidates associated with the anagram value
- ▶ we compute all deletions within a certain distance (e.g. by removing any 2 characters).
 - ▶ Example with 1 character:

$$del(AV(a, e, s, t)) = [AV(a, e, s), AV(e, s, t), AV(a, s, t), AV(a, e, t)]$$

- ▶ This is an arithmetic operation on the anagram values (division)

Variant matching: Search (3)

- ▶ For all of the anagram values resulting from these deletions we look which anagram values in our index *match or contain* the value under consideration. We again gather the candidates that result from all matches.

- ▶ Match or contain: AV_a contains AV_b when

$$AV_a \bmod AV_b = 0$$

- ▶ To facilitate this lookup, we make use of the *secondary index*
 - ▶ Uses a binary search to find the anagrams that we should check our anagram value against (i.e. to check whether it is a subset of the anagram)
 - ▶ Prevents needing to exhaustively try all anagram values in our index.

Variant matching: Search, scoring and ranking (4)

After collecting applying the heuristic and collecting variants, reduce using more conventional means:

- ▶ We compute several similarity metrics between the input and the possible variants:
 - ▶ Damerau-Levenshtein
 - ▶ Longest common substring
 - ▶ Longest common prefix/suffix
 - ▶ Casing difference
- ▶ A score is computed that is a weighted linear combination of the above components
 - ▶ the actual weights are configurable.
 - ▶ an exact match always has score 1.0.
 - ▶ most score components are expressed as a fraction of the input length
- ▶ Frequency as an extra component
- ▶ Optionally, if a confusable list was provided, we adjust the score for known confusables

Parameters and weights

- ▶ Various parameters can be absolute and or relative to the pattern length:
 - ▶ Anagram distance
 - ▶ Edit distance (Damerau-Levenshtein)
 - ▶ Substring length
- ▶ Score threshold
- ▶ Cut-off threshold
- ▶ Max number of matches
- ▶ Weights: determines the importance of a component in the score function
 - ▶ Frequency ranking

Feature: Confusable lists

- ▶ A list of *confusable patterns* with a weight
- ▶ Allows favouring or penalizing certain edits
- ▶ Example: OCR pattern: $-[f]+[s]$
- ▶ Example: historical dutch pattern: $-[uy]+[ui]$ (huys \rightarrow huis)
- ▶ Allows context matching
- ▶ Taken into account as part of the similarity score function

Input and output

Analiticcl takes simple TSV files (tab separated values) as input:

- ▶ **Lexicon**

- ▶ List of preferably validated words/multi-word expressions
- ▶ May contain frequency information

- ▶ **Variant list:** explicitly relates variants to preferred forms.

- ▶ Each variant carries a score expression how likely the variant maps to the preferred word
- ▶ May also contain frequency information
- ▶ **Error list;** a form of a variant lists where the variants are considered errors
- ▶ Example: `separate seperate 1.0 seperete 1.0`
- ▶ This is also the output form in *learn* mode

- ▶ **Language model:** for context-sensitive error detection/correction

- ▶ *Multiple* lexicons/variants lists supported
- ▶ Output is TSV or JSON

Background lexicon

- ▶ Analiticcl depends greatly on the quality of your input (lexicons)
- ▶ A good background corpus is required (out of vocabulary problem)
 - ▶ including morphological variants
- ▶ ..otherwise analiticcl will eagerly mismatch to words it does know!
- ▶ Lexicon may also consist of phrases: less sensitive to false positives
 - ▶ cf. Fuzzy-Search (Marijn Koolen)

Learn Mode (1)

- ▶ Allows extending an existing lexicon with variants
 - ▶ Multiple iterations, covering larger edit distances
- ▶ Outputs a variant list
 - ▶ Can subsequently be used as input again
 - ▶ Possibly after manual curation

Learn Mode (2): Output example

```
{ "Amsterdam": [  
  { "text": "Amsteldam", "score": 0.7499999999999999, "freq": 1 },  
  { "text": "amsterdam", "score": 0.875, "freq": 1 },  
  { "text": "Amster", "score": 0.625, "freq": 1 },  
  { "text": "Amstelredam", "score": 0.6818181818181818, "freq": 1 },  
  { "text": "Amsterd", "score": 0.7321428571428572, "freq": 1 },  
  { "text": "Amstedam", "score": 0.765625, "freq": 1 },  
  { "text": "sterdam", "score": 0.6071428571428572, "freq": 1 },  
  { "text": "Amsterdm", "score": 0.796875, "freq": 1 },  
  { "text": "Tamsterdam", "score": 0.8, "freq": 1 },  
  { "text": "tamsterdam", "score": 0.675, "freq": 1 },  
  { "text": "Amstelredame", "score": 0.6041666666666666, "freq": 1 },  
  { "text": "t'Amsterdam", "score": 0.6136363636363636, "freq": 1 },  
  { "text": "Asterdam", "score": 0.796875, "freq": 1 },  
  { "text": "terdam", "score": 0.5, "freq": 1 },  
  { "text": "Amstelredamm", "score": 0.6145833333333333, "freq": 1 },  
  { "text": "t'amsterdam", "score": 0.6136363636363636, "freq": 1 },  
  { "text": "Amterdam", "score": 0.78125, "freq": 1 },  
  { "text": "msterdam", "score": 0.6875, "freq": 1 },  
  { "text": "amsterd", "score": 0.6071428571428572, "freq": 1 },  
  { "text": "tAmsterdam", "score": 0.675, "freq": 1 },  
  { "text": "amsteldam", "score": 0.6249999999999999, "freq": 1 },  
  { "text": "Amstrdam", "score": 0.75, "freq": 1 }  
]
```

Error Detection (1)

- ▶ In **Query** mode, input is a word/phrase you want to correct as a whole
- ▶ In **Search** mode, input is running text: analiticcl detects which parts of the input (words or higher order n-grams) need to be corrected.
- ▶ An additional and complex challenge!
- ▶ **N-grams**: consider splits and merges:
 - ▶ *thehouse* → *the house* , *teahouse* ?
 - ▶ *tea house* → *the house* , *teahouse* ?
- ▶ **Context** is often a determining factor

Error Detection (2)

Given an input sentence:

1. Extract all segments of the input, i.e. all n-grams up until a certain order
2. Do variant lookup for each (like query mode)
3. Express all segments, their variants, their scores as transitions in a Finite State Transducer (FST)
 - Scores are expressed as costs
4. Extract the best path (lowest cost) with a beam search

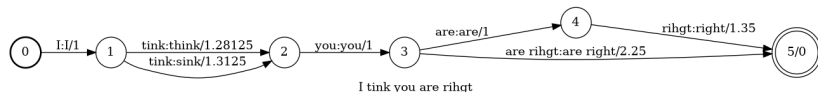


Figure 1: FST

Error Detection (3)

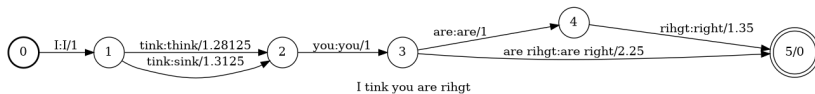


Figure 2: FST

- ▶ Scores are re-expressed as a cost (to be minimised)
- ▶ Base cost (integer) covers the number of input tokens spanned
 - ▶ establish a common ground for comparison between n-grams
 - ▶ n-grams compete
- ▶ Variant cost (fraction): inverse of the variant score: (0.0 best, approaching 1.0 as scores get worse)

$$cost = 1 - score$$

- ▶ **Joint variant score:** Sum of all costs on a complete path.
- ▶ Extract the 'cheapest' path(s)

Error Detection (4): Context

1. Extract the best n solutions from the FST (e.g. $n = 250$)
2. Compute the perplexity for each; using **Language Model**
3. Compute a weighted combined score of the perplexity and the joint variant score
 - ▶ Not trivial, strikes a balance between LM and variant model
 - ▶ Compute normalised joint variant score:

$$variantscore_i = \ln\left(\frac{cost_{best}}{cost_i}\right)$$

- ▶ Compute normalised LM score:

$$lmscore_i = \ln\left(\frac{PP_{best}}{PP_i}\right)$$

- ▶ Weighted geometric mean:

$$score_i = \frac{\lambda_1 variantscore_i + \lambda_2 lmscore_i}{\lambda_1 \lambda_2}$$

4. Select the best scoring solution (minimize score)

Error Detection (5): Output example

het	0:3	het	1
is	4:6	is	0.9879325407796102
een	7:10	een	0.953746422299111 \
		en	0.7713267631867726
huys	11:15	huis	0.8893535359305973 \
		huls	0.7993278799180914

Future Work

- ▶ Evaluation
 - ▶ Golden Agents
 - ▶ Spelling correction task?
- ▶ Comparative study
 - ▶ Analiticcl
 - ▶ TICCL
 - ▶ Fuzzy-Search
- ▶ **Questions?**

References

Software:

- ▶ Analiticcl: <https://github.com/proycon/analiticcl>
- ▶ TICCLtools: <https://github.com/LanguageMachines/ticcltools>
- ▶ Fuzzy-Search: <https://github.com/marijnkoolen/fuzzy-search>
- ▶ Golden Agents NER pipeline: <https://github.com/knaw-huc/golden-agents-htr/tree/master/package>

Publications:

- ▶ Reynaert, Martin. (2004) Text induced spelling correction. In: Proceedings COLING 2004, Geneva (2004).
<https://doi.org/10.3115/1220355.1220475>
- ▶ Reynaert, Martin. (2011) Character confusion versus focus word-based correction of spelling and OCR variants in corpora. IJDAR 14, 173–187 (2011).
<https://doi.org/10.1007/s10032-010-0133-5>