Jamcode 4

A syntax and Python script for syntax-sensitive, context-disambiguated, dictionary-based textual analysis.

Author:

Christian Baden, The Hebrew University of Jerusalem

Recommended Attribution:

Baden, C. (2023). *Jamcode*. Available Online: https://github.com/christianbaden/jamcode

This syntax and script was created as part of the **INFOCORE research project**, coordinated by *Romy Fröhlich, Ludwig Maximilian University Munich*; as part of the work of the **Methodological Working Group: Content Analysis**, headed by *Christian Baden, The Hebrew University of Jerusalem;* which was a joint effort of *Work Package 5: Social Media*, headed by *Dimitra Dimitrakopoulou, Massachusetts Institute of Technology*; **Work Package 6: Strategic Communication**, headed by *Romy Fröhlich, Ludwig Maximilian University Munich*; **Work Package 7: Journalistic Transformation**, headed by *Keren Tenenboim-Weinblatt, The Hebrew University of Jerusalem*; and **Work Package 8: Parliamentary Discourse**, headed by *Rosa Berganza Conde, King Juan Carlos University Madrid.*

The jamcode syntax and Python script include…

- A syntax for context-disambiguated, dictionary-based textual analysis
- A syntax-sensitive procedure for modeling and processing text
- A range of clean-up procedures for multilingual text
- A pipeline for automatically applying context-disambiguated dictionaries to document collections, built for use in conjunction with the AmCAT open source text analysis platform
- Output options as a list of localized matches; term-document matrix; annotated text; and keywords-in-context

The jamcode library contains the following functions:

- gettexts – retrieves document collections from an AmCAT4 server, using the amcat4py API
- towords – cleans up the retrieved text, models its syntax, and represents it as word sequence
- importdict – reads in dictionaries created according to the Jamcode syntax
- jcode (and jcode_ha, for Hebrew/Arabic) – applies imported dictionaries to the text to classify and localize any concepts defined therein within the text
- e_annotate – creates an output file that annotates and labels any concepts within the original text
- e_replace – creates an output file that represents the text as a sequence of recognized concept codes
- e_kwic – creates an output file that shows recognized concepts within the context of adjacent words

In addition, the jcode script governs the sequence of retrieving and processing textual data from an AmCAT4 server, reading in and applying the dictionary, and creating a selection of output files.

I. Purpose & Application

The jamcode syntax and script were developed in order to permit the construction and application of context-disambiguated dictionaries to the study of textual data in different languages. There were several key considerations that drove the development of jamcode:

- **Context matters:** Keyword lists are often insufficient to permit a valid measurement of concepts in text. The same keyword can express quite different meaning depending on context, be that due to polysemy (e.g., 'general' may refer to a military official, but also to a quality of observations; some languages – notably, non-alphabetic ones – contain numerous homographs that need to be disambiguated); due to generic context (e.g., many sentiment dictionaries list 'funny' as an indicator of both positive and negative sentiment, as the term can do both – but in any given use, it is usually only one of these; 'pig' may be an insult in one context, but in another it just refers to a specific animal); or even just because of adjacent qualification (e.g., 'not good' is very different from 'good');

  ▸ **jamcode supports disambiguating keywords using context-based disambiguation criteria.**

- **Syntax matters:** The co-presence of keywords in the same document is usually an insufficient criterion to draw conclusions about their semantic relatedness. To gain additional confidence, it is helpful to consider syntactic properties: Are two words present in the same paragraph, sentence, or clause? How far apart are they in the text?

  ▸ **jamcode considers word distances in a syntax-sensitive fashion**

  In some languages, truncation is complex: Especially in morphologically rich languages, stems are insufficient to reliably recognize all of, and only, valid references to a concept;

  ▸ **jamcode offers a separate coding routine that anticipates common prefixes and suffixes in Arabic and Hebrew language**

  ▸ **jamcode supports restricting truncation to exclude specific characters**

- **Character encoding matters:** In many languages, there are subtly different ways of spelling the same words, which need to be harmonized for valid measurement (e.g., using diacritics; special characters such as 'ß' in German that can be replaced by 'ss'; in some languages, such as Serbian, different alphabets are used alongside one another);

  ▸ **jamcode includes some** (admittedly quite manual) **harmonization routines for character encoding**

Beyond this, the jamcode script and syntax anticipates several different uses of automated textual analysis that require different data formats and representations. Specifically, jamcode supports the following output formats:

- **Localized concepts** – a list of recognized matches, identifying the document of origin and word position;[1] this is the base format from which all other formats are derived;
- **Document-term-matrix** – a table that counts, for each document and each concept, how often a concept was found; this format is needed for subsequent bag-of-words type or frequency-based analyses;
- **Annotated text** – an annotated version of the original text, with concept labels attached to every matched word; this format is useful for validating dictionaries and subsequent qualitative analyses;

[1] jamcode uses a word position counter that a) distinguishes between the title, subtitle, and body of a document, and responds to syntax, i.e., the counter does not strictly represent the how-many-th word in a text was annotated, but 'how far' from the origin a concept was found, factoring in clause, sentence, and paragraph breaks as part of the distance.

- **Concept code sequence** – a representation that lists, in sequence, all concept codes that have been applied to a text; this format can be used for subsequent topic modeling, departing from a level of recognized concepts, not words; it is also useful for validating whether the overall meaning of a document is captured by the recognized concepts
- **Keywords-in-context** – an output file that lists all words matched based on a dictionary, surrounded by a number of preceding and succeeding words; this format is useful for identifying suitable disambiguation criteria, and can be used for subsequent concordance analyses

II. Syntax

The jamcode syntax works as follows.

Any concept is operationalized using one or multiple search phrases. Search phrases are separated by space: Everything that is between two spaces is considered to be part of the same search phrase, and everything that is separated by spaces is considered to be part of distinct search phrases.

Please note:
Unnecessary spaces are the most common source of error. Under all circumstances, avoid double spaces, since this creates an empty search phrase (everything between two spaces) that will match any location in a document.

Every search phrase begins with a keyword. A keyword can be any character string that consists of letters in any recognized alphabet or script, numbers, or emojis. Punctuation marks cannot be part of keywords, and also selected special characters are excluded (notably, these: ()_&|*~). Keywords can be truncated, using * as truncation mark. Truncations can be at the beginning ("*abc" will match "abc", but also "xyzabc") and/or at the end ("abc*" will match "abc", but also "abcdef") of a keyword, but not in between ("a*c" will not work). If no truncation is defined, keywords match only character sequences enclosed by spaces (e.g., "abc" will not match "abcd", but "abc*" will; the exception to this rule is that in Hebrew and Arabic, certain common prefixes and suffixes will be automatically included, see below).

To use additional disambiguation criteria, the keyword is followed (without intermittent space) by an underscore _, an identifier of the type of disambiguation criterion (t,y,n,p,s), and a bracket that defines the criterion (e.g., "abc*_y(…)". The same keyword can be amended by any number of disambiguation criteria. The keyword will be matched only if all disambiguation criteria are fulfilled (e.g., "*abc_y(…)_n(…)_n(…)" will be matched only if all three criteria are met).

Please note:
Disambiguation criteria are optional. Everything within each disambiguation criterion must be entered without any spaces (everything separated by spaces will be considered to be part of a different search phrase). Multiple criteria are concatenated using underscores.

There are five types of disambiguation criteria, which fall into three classes: temporal criteria, context words, and affixes.

1. *Temporal criteria*

Keywords can be disambiguated based on document metadata, notably, the registered publication date. This permits, for instance, coding office holders only while they hold office (so "Trump" is coded as a reference to "US President" only while he is in office, but not before or afterwards).

_t(…) This type of criteria defines the date from and until when a keyword is considered.

Temporal criteria are specified as dd/mm/yy-dd/mm/yy, using double digits for single-digit days, months and years; years are assumed to be in the 2000s, i.e., _t(03/07/14-10/07/14) refers to 3-10 July 2014. Start date and end date can be identical (so only one day will be considered), but the end date cannot be before the start date.

Please note:
Unlike the date criterion in the dictionary line (see below), this criterion governs only the use of a specific search phrase, while all other search phrases for the same concept will still be applied.

2. Context words

Keywords can be disambiguated based on whether other character sequences are found within a given word distance.

_y(…) This type of criteria defines which additional character sequences *must* be found for a keyword to be matched.

_n(…) This type of criteria defines which additional character sequences *must not* be found for a keyword to be matched.

Context words criteria contain two types of information: The character sequences that must or must not be found; and the word distance wherein the criterion applies. Character sequences are listed at the beginning of the bracket, using the following Boolean operators:

&	AND
|	OR
()	brackets

Character sequences are defined in the same way as keywords, i.e., they can contain letters, numbers, and emojis, and can use truncation, but cannot include punctuation or excluded special characters.

The word distance is defined at the end of the bracket, by a number preceded by ~. Word distance is symmetric, i.e., ~5 denotes a word distance of up to 5 words before or after a detected keyword. Word distance automatically recognizes the beginning or end of documents (so if the document ends within the specified distance, only existing words are considered)

Please note:
Within the jamcode *coding script, punctuation is automatically converted into word distance: Minor syntactic breaks (commas, semicola, dashes) are counted as one word; sentence breaks (periods, question marks, exclamation points) are counted as three words; and paragraph breaks are counted as five words. Thus, in the example "abc de fgh i. jkl, mno pq.", the distance between "abc" and "pq" is counted as 10 words (three words, three for the period, one word, one for the comma, two words).*

For instance, the disambiguation criterion _y(abc|def*~5) is matched if either "abc" or "def*" (or both) are found between five words before and five words after the detected keyword; the disambiguation criterion _n(*abc*|(def&gh)~10) is matched if the ten words before and after the detected keyword include neither a word that contains the "abc", nor both of the sequences "def" and "gh"; if said words contain only "def", the criterion is not violated (it excludes only the joint use of "def" and "gh"), so the keyword would be matched.

Please note:
Each layer of brackets must contain only operands of the same type; for instance, (a|b|(c&d)|e~5) is consistent, as is (a&(b|(c&d))~10) or ((a&b)|(c&d)|(e|f)), although the last bracket around "e" and "f" is unnecessary; By contrast,

(a&b|c~15) is inconsistent. Whenever operands of different kinds coexist in a criterion, they need to be ordered by consistent brackets;
A bracket within the criterion must always include at least two elements; _y(abc~2) is consistent, but _y((abc)~2) is not, nor is _y(a|b|(c)~2);
The current coding script supports only up to five levels of nested brackets; Unbalanced brackets will raise an error.

3. *Affixes*

Keywords can furthermore be disambiguated based on what prefixes and suffixes are permitted. This type of criteria is relevant under two conditions:

1) for truncated keywords, it excludes specific characters in the truncation;

For truncated keywords, it is sometimes useful to exclude specific characters, so the keyword can be preceded or followed by any character except for those specified (e.g., "aid*" also matches "AIDS", but if the suffix "s" is excluded, it does not)). The character position checked for consistency with this criterion is the one immediately adjacent to the character sequence without the truncation.

2) for Arabic and Hebrew language;

Arabic and Hebrew both tend to affix various function words to keywords ("בית" is "house" and "הבית" is "the house", so the first character is a prefix of a lexical unit). In addition, there are specific endings for gendered forms, which are highly regular and can be treated automatically. Therefore, the jamcode coding routine for Arabic and Hebrew defines a range of common prefixes and suffixes in each language, which are automatically considered (i.e., the keyword "בית" automatically matches also "הבית" or "ביתך" even if no truncation is defined). Specifically, the following affixes are pre-defined as part of the coding routine:

Arabic Prefixes	بال ٱفال ٱلل ال ل ن ي ت ل و ب ك م ف
Arabic Suffixes	ا ون نا تن تم ت ن وا ي كم ك ه هم ها كن ة ية ين ان
Hebrew Prefixes	מה נ ת י א ש ה מ ל כ ב ו
Hebrew Suffixes	ית ם ן ך ת תן תם תי ה כן כם הן הם נו ו כ י ות ים

Under normal circumstances, automatically permitting these prefixes and suffixes increases coding accuracy; however, every now and then, specific affixes create ambiguities, notably, if a word plus a specific prefix or suffix forms a homograph with another word that also exists.

In such cases, the affix criteria serve to expressly *exclude* specific affixes:

_p(…)	This type of criterion excludes a specific prefix, so the keyword will not be matched if the character sequence is preceded by any listed character
_s(…)	This type of criterion excludes a specific suffix, so the keyword will not be matched if the character sequence is followed by any listed character

For instance, _p(מ) is satisfied if the final character before a recognized keyword is either a space or any of the listed prefixes, but *not* "מ". Likewise, _s(ה) is satisfied if the first character following the recognized keyword is either a space or any of the listed suffixes, but *not* "ה".

III. Dictionary files

A dictionary for use in jamcode can contain any number of concepts.

Each line defines a separate concept.

Every line follows a strict format of exactly four fields:

1. *Concept identifier*
   The concept identifier is a number that is unique to this concept, and will be written into most output formats, for efficient data handling.

2. *Concept label*
   The concept label can contain any free text, but neither tabs or line breaks. This label will be used as column head in the extended term document matrix format and in the annotated output format.

3. *Date criterion (optional)*
   Every concept can be restricted in time, i.e., the concept will only be coded in documents dated within a specified date range. Temporal criteria are specified as dd/mm/yy-dd/mm/yy, using double digits for single-digit days, months and years; years are assumed to be in the 2000s, i.e., 03/07/14 refers to 3 July 2014. Start date and end date can be identical (so only one day will be considered), but the end date cannot be before the start date. If the date criterion is empty, the concept will be coded regardless of the date.

   Please note:
   Unlike the disambiguation criterion (see above), this field governs the coding of the entire concept, and will be applied to every search phrase listed thereunder.

4. *Search phrases*
   This field contains all search phrases as defined above. There has to be at least one keyword, but there is no restriction to the number of keywords or keywords-plus-disambiguation-criteria search phrases permitted. All unique search phrases are separated by spaces. There is no particular order to these search phrases. A concept will be coded if the text matches any included search phrases (i.e., if the same word in a document matches multiple search phrases, it will still be coded only once).

   Please note:
   *To prevent redundant coding of adjacent words due to multiple matched search phrases (e.g., "not necessary" matches both "not_y(needed|necessary|important~5)" (the code will be applied to the keyword, "not"), and "neces*_y(no|none|neither|nor|never~10)" (the code will be applied to the keyword, "neces*"), the* jamcode *script includes a restriction that the same concept code will not be applied again within a word distance of 5 from an initial recognized position: If multiple search phrases identify the same concept in a multiword expression, the first word that matches a search phrase will be marked, and the subsequent ones will be ignored). This exclusion is lifted if another concept is coded in another word (i.e., if the first word is coded as concept A, the second is coded as concept B, then the third can again be coded as concept A). This exclusion is useful if the dictionary contains many multi-word expressions, and there are many coded concepts, so distinct nearby references to the same concept are usually separated by other coded concepts. For applications where this is not the case, the easiest way to circumvent this restriction is to add a final concept to the dictionary with * as the search phrase, which thus matches any non-space, non-punctuation character sequence in the document; this ensures that also nearby mentions of the same concept will always be separated (by this added concept), and thus correctly annotated.*
   Make sure that there are no unnecessary spaces in the search phrases field, i.e., neither double spaces, nor initial spaces, nor final spaces, nor any spaces within a search phrase.

The dictionary is passed to the jamcode coding script as a plain text file, named "DICT_<name>_<language>.txt", where <name> is a label for the dictionary, which should only contain letters or numbers, but no spaces, and <language> is a two-character, upper-case indicator of the dictionary's language (at present, jamcode recognizes

the following languages: Albanian (AL), Arabic (AR), German (DE), English (EN), French (FR), Hebrew (HE), Macedonian (MA), Serbian (SR); if jamcode does not recognize a language, it will be treated like English, i.e., with minimal pre-processing)

Specifying language is important for two reasons.

1) If the language is recognized by the jamcode script, it will apply several language-sensitive harmonization scripts, which serve to efficiently handle special characters and diacritics. For instance, for Serbian, which uses either the Roman alphabet or a variant of the Cyrillic one, all text will be converted into a common character space; the jamcode script includes routines for handling language-specific ways of expressing acronums (e.g., the interspersed " in Hebrew), and other cleanup tools. If the language is not recognized by the script, it will be treated like English, i.e., with minimal intervention.
2) Identifying the language as Arabic (AR) or Hebrew (HE) switches on the pre-defined prefixes and affixes in the coding process (see above).

In the dictionary text file, every line defines one coded concept. There is no header line. Fields are separated by tabs. A dictionary has to contain at least one line. There cannot be any empty lines.

For instance, a minimal dictionary could look as follows (note the empty third, date criterion field):

1	Concept A	abc def_y(gh~2)
2	Concept B	ij kl_n(mn~5)_t(10/07/02-17/07/02)

IV. jamcode functions

gettexts(<index>,<fromnr>)

gettexts obtains an ordered list of text and metadata from an AmCAT4 server. To do so, it connects to the AmCAT4 API, amcat4py, which must be installed and loaded; the address of the relevant AmCAT4 server must be specified in the script (line 61), and the user must be logged onto that server on the relevant computer (for details on how to use the AmCAT4 API, please refer to https://amcat.nl).

<index> the index name assigned to a document collection on AmCAT

<fromnr> gettexts can obtain only documents starting from a given document ID number; this can be useful if a prior request crashed before completing. If <fromnr> is left unspecified, it defaults to 0, i.e., all documents in the collection will be retrieved.

By default, gettexts obtains the data fields headline, byline, text, as well as the metadata fields _id (or id), date, and medium from the server. Textual data is encoded in Unicode.

towords(<text>,<language>)

towords transforms a Unicode-encoded text string (e.g., the title, subtitle, or text data obtained by gettexts) into a tuple of words.

<text> any Unicode-encoded text string

<language> a two-character language identifier; at present, jamcode recognizes Albanian (AL), Arabic (AR), German (DE), English (EN), French (FR), Hebrew (HE), Macedonian (MA), Serbian (SR); if jamcode does not recognize a language, it will be treated like English, i.e., with minimal pre-processing)

As part of the preprocessing, towords cleans up many special characters.

Cyrillic Serbian or Macedonian is transliterated into Roman characters, in order to harmonize uses in different scripts, and Arabic numerals (١,٢,٣) are converted into Romanized Arabic numerals (1,2,3). importdict does the same for dictionary entries, so a dictionary for Serbian or Macedonian can be specified in either Cyrillic or Roman characters, but will be applied consistently to both Cyrillic and Roman text.

towords also recognizes sentence and paragraph syntax and replaces it by words. This serves to introduce a penalty toward the coding algorithm: Additional words specified in the Boolean queries must be closer if they are located in different syntactical units than if they are in the same clause, sentence, or paragraph.

- Comma, Colon, Semicolon, Dash and Quotation Marks are treated as one word each.
- Period, Exclamation Mark, and Question Mark are treated as three words each.
- Paragraph breaks are treated as five words each.

This excludes punctuation that is not used as punctuation: For instance, periods within numbers are preserved; In Arabic and Hebrew, acronyms using " or ' are preserved. In English and French, in-word apostrophes are replaced by spaces, so 'I'm' becomes 'I m', or ['I', 'm'] as tuple.

importdict(<dictionary>)

importdict imports a dictionary file named DICT_<dictionary>.txt, using the jamcode syntax (see above).

<dictionary> the name of the dictionary, as part of the file name DICT_<dictionary>.txt; names should end with a two-character language identifier as specified above (e.g., 'test_EN', which would indicate an English language dictionary)

importdict applies the same script and special character conversion routines as towords.

jcode(<words>,<dict>,<date>,<adjacent=0>)

jcode is the main coding script, which applies the criteria laid down in the dictionary read by importdict to the tuples of words obtained from towords. It returns a list of all found concepts, and their word position within the text.

<words> a list of words

<dict> the dictionary read-in by importdict

<date> the date associated with the coded text

<adjacent> an option that defines whether adjacent codes permitted: By default, jcode only records successive instances of the same code if these are separated either by at least five words, or a different code. This is to prevent overlapping coding criteria from registering multiple matches in multi-word expressions multiple times. If left unspecified, <adjacent> defaults to 0, i.e., adjacent codes are not permitted. If set to 1, the option switches off this restraint, such that all matching instances are recorded even if they are adjacent.

jcode returns a list of matches, consisting of an identifier of the position within the word list where a matching word was found, and the attributed concept code (e.g., [17, '471'] means that the concept with the concept identifier 471 in the dictionary was matched to the 17th word in the list of <words>).

jcode_ha(<words>,<dict>,<date>,<lang>,<adjacent=0>)

jcode_ha is a variant of the main coding script used for morphologically rich languages – in this case, Arabic and Hebrew – which applies the criteria laid down in the dictionary read by importdict to the tuples of words obtained from towords. It returns a list of all found concepts, and their word position within the text.

<words> a list of words

<dict> the dictionary read-in by importdict

<date> the date associated with the coded text

<lang> a two-character, upper-case language identifier (here, AR for Arabic or HE for Hebrew)

<adjacent> an option that defines whether adjacent codes permitted (see above)

As morphologically rich languages, Hebrew and Arabic use prefixes to express many conjunctions and articles, so the algorithm allows for certain prefixes to appear before the keyword. It also allows specific regular suffixes. For this purpose, jcode_ha contains several lists of common prefixes and suffixes:

mypref1 one-character prefixes

mypref2 multi-character prefixes

mysuf suffixes

These lists are, however, potentially overridden by any prefix and suffix exclusion criteria specified in the dictionary (_p() and _s() criteria, see above).

As a consequence, a keyword specified in the dictionary will match not only the exact keyword as specified, but also the same keyword plus any of the permitted prefixes or suffixes.

Other than that, jcode_ha functions the same as jcode.

e_annotate(<words>,<found>,<dict>

e_annotate re-combines the tuple of words obtained from towords, the list of recognized concepts obtained from jcode or jcode_ha, and the concept names and ids obtained from importdict, to return a string wherein each recognized word is appended a bracket specifying which concept has been recognized.

<words> a list of words

<found> the list of matches returned by jcode or jcode_ha

<dict> the dictionary read-in by importdict

Example:

Raw text: 'I think, therefore I am confused.'

…processed by towords becomes:

Word tuple: ['I','think','xxcom','therefore','I','am', 'confused', 'xxdot', 'xxx', 'xxx']

…when processed with jcode, using this dictionary:

Dictionary:

101	Think	think*_n(pad~2)
102	Myself	I

…returns three hits:

Matches: [[0,'102'],[1,'101'],[4,'102']]

…which will be re-combined by e_annotate as follows:

Output: 'i(Myself) think(Think), therefore i(Myself) am confused.'

e_replace(<found>)

e_replace uses the list of recognized in a text to simply list all recognized concept ids, separated by spaces.

<found> the list of matches returned by jcode or jcode_ha

Example:

Raw text: 'I think, therefore I am confused.'

processed in the same way as in the above example, will be represented by e_replace as follows:

Output: '102 101 102'

e_kwic(<hit>,<b>,<words>,<dict>)

e_kwic uses the tuple of words obtained from towords, a single match obtained from jcode or jcode_ha, and the concept names and ids obtained from importdict to return a sequence of string of the recognized keyword in context.

<hit> a single match returned by jcode or jcode_ha

<b> a bandwidth parameter, specifying how many words before and after a match are shown

<words> a list of words

<dict> the dictionary read-in by importdict

e_kwic creates an output object structured as follows:

concept identifier, concept name, up to <b> words preceding the matched word, matched word, up to <b> words succeeding the matched word

Example:

Raw text: 'I think, therefore I am confused.'

processed in the same way as in the above example, using a bandwidth of 3, will be represented by e_kwic as follows:

Output Match 1:	'102',	'Myself',	'',	'i',	'think, therefore'
Output Match 2:	'101',	'Think',	'i',	'think',	', therefore i'
Output Match 3:	'102',	'Myself',	'think, therefore',	'i',	'am confused. '

Please Note: the comma counts as one word

V. The jcode script

jcode.py is the governing script that executes the coding pipeline: It…

1) uses gettexts to access an AmCAT server and obtain documents, which are preprocessed using towords;
2) loads one or multiple dictionaries called DICT_<dictionary>.txt;
3) codes the obtained texts according to the dictionaries, and
4) exports the recognized concepts into a results file, and several optional additional files.

jcode.py is called as follows:

jcode.py <index> <dictionary> <options>

<index> the index name assigned to a document collection on AmCAT

<dictionary> the name of the dictionary, as part of the file name DICT_<dictionary>.txt; names should end with a two-character language identifier as specified above.[2]

Alternatively, you can also enter INDEX as <dictionary>; in this case, jcode will search for a file called lang_index_<index>.csv in the python folder, wherein each line <document id>,<dictionary> specifies which out of multiple dictionaries is to be used for each document.

<options> multiple additional, optional functions, namely:

- s creates a simple document term matrix as output file, named td_<index>_<dictionary>.txt. In this matrix, the column headers contain the concept ids, and row headers contain the document ids.
- e creates an extended document term matrix as output file, named td_<index>_<dictionary>.txt. In this matrix, the first two rows of column headers contain 1) the concept ids and 2) the concept names; and the first three columns state of row headers contain 1) the document ids, 2) the date, and 3) the medium.
- a creates an annotated text file as an output file, named annotated_<index>_<dictionary>.txt, which contains all documents annotated with concept labels attached to every matched word.

  Please Note: this option considerably slows down the script and creates rather large files if applied to big document collections.
- r creates an output file called replaced_<index>_<dictionary>.txt, wherein each document is represented by the sequence of all concept ids recognized in this document
- k creates a keywords-in-context list as output file named kwic_<index>_<dictionary>.txt, which lists, for each concept, all found instances within their original context

  Please Note: By default, this option includes up to five words before and after a coded concept. If the option k is followed by a space and a number, that word distance will be used instead (e.g., 'k 10' will use a word distance of 10).
- j permits the coding of adjacent matches: By default, jcode only records successive instances of the same code if these are separated either by at least five words, or a different code. This is to prevent overlapping coding criteria from registering multiple matches in multi-word

[2] <dictionary> can be any name given to the dictionary file; however, as jamcode already includes several cleanup procedures for text in English (EN), French (FR), German (DE), Albanian (AL), Serbian (SR), Macedonian (MA), Arabic (AR) and Hebrew (HE), these procedures are only applied if the dictionary name ends with any of the relevant, upper-case two-character language identifiers. If <dictionary> ends in any other way, only minimal preprocessing will be applied.

expressions multiple times. The option switches off this restraint, such that all matching instances are recorded even if they are adjacent.

from<document id> commences the coding not from the first document in the set, but the first with an id equal to or larger than the specified number.

Jcode always generates an output file named results_<index>_<dictionary name>.txt, which lists, for every document, and in sequence, every matched concept along with the matched word position

Output: document id, word position, concept id

wherein:

document id is read from AmCAT;

word positions are prefixed 't' for title, 's' for subtitle/byline, and 'a' for article; word position responds to syntax, i.e., its value indicates the position of a matched word after counting clause breaks (commas, colons, semicolons, dashes, quotation marks) as one, sentence breaks (periods, exclamation points, question marks) as three, and paragraph breaks as five words;

concept id is read from the dictionary.

VI. License & Disclaimer

Please Note Also:

jamcode is scientific software written by a non-expert, autodidactic coder (i.e., me), over an extended period of time. The original version was written for Python 2.7.3, and numerous edits have been made to add functions, adjust the code to Python 3.x and AmCAT 4, and for various other purposes. I am fully aware that there is a good chance that there are still bugs in the software, and I have no doubt that expert code developers will find my code inefficient, ugly, and intransparent. That's just how it is. If you want to clean up my code, make it more efficient, or adapt it to use more powerful libraries and packages, go ahead, and feel free to let me know – but please credit the original work. It may not be pretty, but it works, and I believe it achieves something that is useful, new, and valuable, and it was a lot of work to get there.