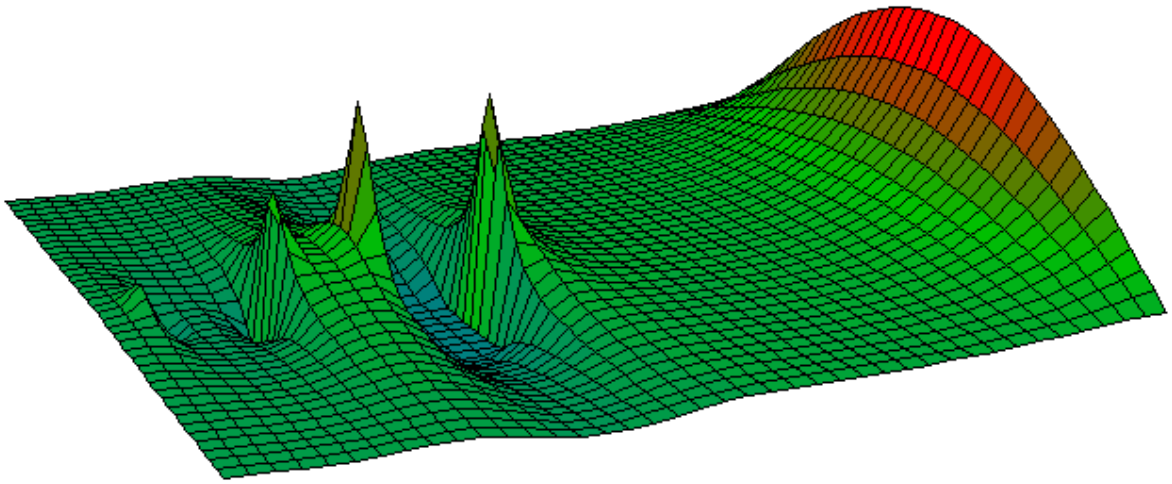
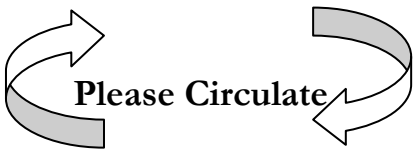
Journal of J

An interdisciplinary journal on J programming language and applications in science





MPM press

A open access Journal

ISSN: 2174-9280

Vol.5, No.1 August 2017

Document Similarity using Tf-Idf Model - the J way

Martin Saurer / May 2017

martin DOT saurer AT bluewin DOT ch

During the last months, I was involved with a research project, where a large part was related to text analysis and data mining. One of our goals was to find similar documents corresponding to different sets of keywords. For this project, we decided to use Python [1] as the programming language, and gensim [2] as one of many libraries. Python was a good choice for that, so all team members could easily understand the program code.

Many hundred lines of Python code later, I wondered whether some algorithms, currently implemented in Python, could be designed more efficient and elegant in another programming language. Python is a fantastic programming language, but it was not built to handle large arrays as one of its core functionalities.

Large arrays and matrices are core elements in text analysis algorithms, so why not using an array oriented programming language? A programming language which is able to handle arrays as easy as they were single values. This paper explains the implementation of the Tf-Idf (Term frequency - Inverse document frequency) model, as well as the calculation of the "cosine similarity" score in J [3].

Tf-Idf / cosine similarity is a widely used technique in machine learning, information retrieval and text mining. The mathematical background of this method can be found in the so called "Vector Space Model" (VSM) which is an algebraic model, representing words as vectors.

For example, take a look at a todays internet search engine. It takes your input, which normally contains the keywords of what you are looking for, and then retrieves a series of documents (hyperlinks) according to your query. The results are in order from the highest to the lowest hit rate. The hit rate is represented by a score-value, calculated using the document words, and the keywords (query). The higher the score-value, the more a retrieved document matches the query.

Tf-Idf / cosine similarity - explained in theory

The following description is extremely short and covers only a small part of the whole theory and background. For a deeper introduction on "Machine Learning / Text Feature Extraction", please see Christian Perone's excellent blog [4] [5] [6], as well as the article in Cambridge University Press about “The vector space model for scoring” [7].

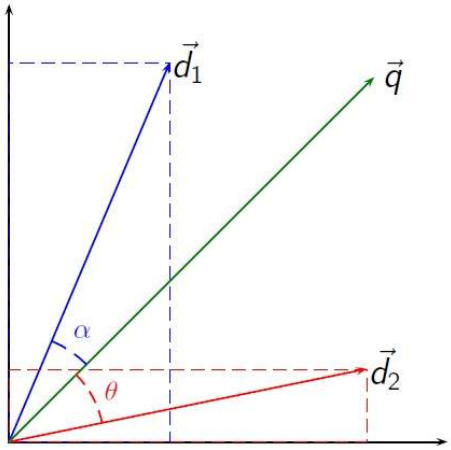
The common form of the Tf-Idf cosine similarity is:

$$similarity\_score = cos\theta = \frac{d \cdot q}{|d| \times |q|} = \frac{\sum_{i=1}^{n} d_i q_i}{\sqrt{\sum_{i=1}^{n} d_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}} = \hat{d} \cdot q$$

where:

d	=	document idf vector
$\hat{d}$	=	normalized document idf vector
q	=	query / normalized idf search bow vector
$\|d\|$	=	Length of d (extension of Pythagoras's theorem)
$\|q\|$	=	Length of q (extension of Pythagoras's theorem)
$\cdot$	=	Dot product of two vectors: $\sum_{i=1}^{n} d_i q_i$
$\times$	=	Product of two numbers

Graphical representation of the "cosine similarity":



Example:

$$similarity_{txt:query} = cos\theta \frac{corpus_{idfs} \cdot query_{normalized}}{|corpus_{idfs}| \times |query_{normalized}|} = 0.3857$$

$$similarity = cos\theta = \frac{[1.0 \quad 0.0 \quad 0.415 \quad 0.415 \quad 0.0 \quad 0.0] \cdot [0.4472 \quad 0.8944 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0]}{|[1.0 \quad 0.0 \quad 0.415 \quad 0.415 \quad 0.0 \quad 0.0]| \times |[0.4472 \quad 0.8944 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0]|}$$

$$= \frac{0.4472}{1.1595 \times 1.0000}$$

$$= 0.3857$$

The way on how to get to the vectors used in this example will be shown later in detail.

Tf-Idf cosine similarity - calculated in gensim:

The gensim Python library uses a little different approach to calculate the cosine similarity using the Tf-Idf model.

$$similarity = cos\theta = \hat{d} \cdot \hat{q}$$

$\hat{d}$ = normalized document idf vector
$\hat{q}$ = normalized idf search bow vector of the query

$\cdot$ = Dot product of two vectors: $\sum_{i=1}^{n} d_i q_i$

Example:

$$similarity_{txt:query} = cos\theta = \hat{corpus}_{idfs} \cdot \hat{query} = 0.3857$$

$$similarity = cos\theta = [0.8624 \quad 0.0 \quad 0.3579 \quad 0.3579 \quad 0.0 \quad 0.0] \cdot [0.4472 \quad 0.8944 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0]$$

$$= 0.3857$$

But enough of boring theory. Let's see how to implement these concepts in J. Please note that the following implementation is NOT usable for real-world scenarios, simply because it's too slow. It demonstrates the calculation of the cosine similarity in detail, for educational purposes. After this educational part, we will take a look at the fully optimized version, which is more difficult to understand, but the calculation part implemented in J is roughly four times (!) faster than Python.

Some pre-requisites:

Before analyzing texts, it's a good idea to perform some pre-processing tasks such as eliminating punctuation characters like commas, periods, etc. After that, we convert the whole text to lowercase, and remove so called stop-words, such as "the", "is", etc. Stop-words are irrelevant for text analysis, because they do not contain any information of interest.

```
punc_chars =: ',;.:'
stop_words =: 'the';'is';'we';'can';'in'
```

Documents (training set):

```
txt_0 =: 'The sky is blue'
txt_1 =: 'The sun is bright'
txt_2 =: 'The sun in the sky is bright'
txt_3 =: 'We can see the shining sun, the bright sun'
txt_n =: txt_0;txt_1;txt_2;txt_3
```

Document to get the similarity score compared to txt_0-3 (test set):

```
search_text =: 'The sky is blue'
```

Question 1: What is the similarity score of search_text compared to txt_0 ?

```
search_text =: 'The sky is blue'
txt_0       =: 'The sky is blue'
```

This one is easy. search_text has a similarity score of 1.0 compared to txt_0, because search_text and txt_0 are the same.

Question 2: What is the similarity score of search_text compared to txt_2 ?

```
search_text =: 'The sky is blue'
txt_2       =: 'The sun in the sky is bright'
```

This one is harder to determine using the naked eye, but it's possible to calculate the similarity score by performing the following steps.

Some utility verbs - or "first things first":

Before we really start the text analysis, let's define some utility verbs.

```
NB. Remove punc_chars from text in y
remove_punc_chars =: 3 : 0
   (I. -. (a. i. y) e. (a. i. punc_chars)) { y
)

NB. Remove stop_words from words in y
remove_stop_words =: 3 : 'stop_words -.~ y'

NB. Get indices of dictionary entries in y
get_dict_entries =: 3 : 'dict_words i. y'

NB. Classify each entry in y
classify_entries =: 3 : 'y ({. ; #) /. y'

NB. Get missing indices
get_missing_indices =: 3 : 'I. -. dict_words e. y'

NB. Prepare missing indices
prepare_missing_indices =: 3 : '|: (y) ,. (<0)'
```

```
NB. Prepare corpus
prepare_corpus =: 3 : '|:(]/:{"1)|:(>0{y),.(>1{y)'

NB. Get idf of a word in txt_n
get_idf =: 3 : '+/ 0 >~ (+/ |: -. (<y) i. >txt_n)'

NB. Normalize a vector
normalize =: 3 : 'y%(%:+/y^2)'
```

Start:

```
]txt_n
┌─────────────────┬──────────────────────┬──────────────────────────┐...
│The sky is blue  │The sun is bright     │The sun in the sky is     │...
└─────────────────┴──────────────────────┴──────────────────────────┘...
┌──────┬──────────────────────────────────────────────┐
│bright│We can see the shining sun, the bright sun    │
└──────┴──────────────────────────────────────────────┘
```

Step 1:

Convert all words to lowercase.

```
NB. Convert all words to lower case
]txt_n =: tolower each txt_n
┌─────────────────┬──────────────────────┬──────────────────────────┐...
│the sky is blue  │the sun is bright     │the sun in the sky is     │...
└─────────────────┴──────────────────────┴──────────────────────────┘...
┌──────┬──────────────────────────────────────────────┐
│bright│we can see the shining sun, the bright sun    │
└──────┴──────────────────────────────────────────────┘
```

Step 2:

Remove punctuation characters.

```
NB. Remove punctuation characters
]txt_n =: remove_punc_chars each txt_n
┌─────────────────┬──────────────────────┬──────────────────────────┐...
│the sky is blue  │the sun is bright     │the sun in the sky is     │...
└─────────────────┴──────────────────────┴──────────────────────────┘...
┌──────┬──────────────────────────────────────────────┐
│bright│we can see the shining sun the bright sun     │
└──────┴──────────────────────────────────────────────┘
```

Step 3:

Split sentences into single words.

```
NB. Split sentences into single words
]txt_n =: cut each txt_n
```

the	sky	is	blue

the	sun	is	bright

the	sun	in	the

...

sky	is	bright

we	can	see	the	shining	sun	the	bright	sun

Step 4:

Remove stop-words.

```
NB. Remove stop words
]txt_n =: remove_stop_words each txt_n
```

sky	blue

sun	bright

sun	sky	bright

...

see	shining	sun	bright	sun

Step 5:

Create a dictionary that contains all words from all documents (without stop-words), and assign a unique id (top row) to each word.

```
NB. Create dictionary (each word gets a unique id)
]dict_words =: (~.(,/>txt_n))-.<''
```

sky	blue	sun	bright	see	shining

```
]dictionary =: |:(;/i.#dict_words),.dict_words
```

0	1	2	3	4	5
sky	blue	sun	bright	see	shining

Step 6:

Build a corpus (bag of words), where txt_0-3 are represented by each words unique id from the dictionary, and the number of how many times a word/term appears in a document (the term frequency).

```
NB. Create corpus (bag of words, ids and number of
NB. occurence in text)
]corpus =: |: each classify_entries each get_dict_entries
each txt_n
```

0 1	2 3	2 0 3	4 5 2 3
1 1	1 1	1 1 1	1 1 2 1

```
]fill_indices =: |: <"0 each > each (get_missing_indices
each txt_n)
```

2 3 4 5	0 1 4 5	1 4 5	0 1

```
]fill_indices =: prepare_missing_indices each fill_indices
```

2 3 4 5	0 1 4 5	1 4 5	0 1
0 0 0 0	0 0 0 0	0 0 0	0 0

Then calculate the "missing" indices, because not all documents contain all words from the dictionary. We fill-up the resulting vectors, so that each document vector contain all dictionary id's plus the corresponding term-frequency. The term-frequency may have a value of 0 (zero) if the dictionary-word does not appear in the document.

```
]corpus =: corpus,.fill_indices
```

0 1 1 1	2 3 4 5 0 0 0 0
2 3 1 1	0 1 4 5 0 0 0 0
2 0 3 1 1 1	1 4 5 0 0 0
4 5 2 3 1 1 2 1	0 1 0 0

And finally we build the corpus as we use it for calculation.

```
]corpus       =: <"2 prepare_corpus"1 corpus
```

0 1 2 3 4 5 1 1 0 0 0 0	0 1 2 3 4 5 0 0 1 1 0 0	0 1 2 3 4 5 1 0 1 1 0 0	0 1 2 3 4 5 0 0 2 1 1 1

Let's have a closer look at the corpus, for example the last cell which corresponds to:

```
txt_3 =: 'We can see the shining sun, the bright sun'
```

Without stop-words and punctuation characters and split into single words:

see	shining	sun	bright	sun

So the meaning of the last corpus cell is as follows:

0	1	2	3	4	5	Dictionary IDs
sky	blue	sun	bright	see	shining	Dictionary Words
0	0	2	1	1	1	Word Frequency

In fact, we only need the "Word Frequency" row. The rest is just for clarification.

Step 7:

Create a bag of words (bow) for the test set (search_list), similar to the corpus from step 6.

```
NB. Prepare search_text
]search_text =: tolower search_text
the sky is blue

]search_text =: remove_punc_chars search_text
the sky is blue

]search_list =: cut search_text
┌───┬───┬──┬────┐
│the│sky│is│blue│
└───┴───┴──┴────┘

]search_list =: remove_stop_words search_list
┌───┬────┐
│sky│blue│
└───┴────┘

NB. Create search bow
]search_bow =: |: classify_entries I. dict_words e.
search_list
┌─┬─┐
│0│1│
├─┼─┤
│1│1│
└─┴─┘

]fill_bow   =: get_missing_indices search_list
2 3 4 5
```

```
]fill_bow   =: |:(<"0 fill_bow),.(#fill_bow)$(<0)
```

2	3	4	5
0	0	0	0

```
]search_bow =: search_bow,.fill_bow
```

0	1	2	3	4	5
1	1	0	0	0	0

```
]search_bow =: |: (]/:{"1) |: search_bow
```

0	1	2	3	4	5
1	1	0	0	0	0

Step 8:

Extract document vectors from corpus, and the search bow vector from search_bow.

```
NB. Extract corpus bow vectors
]corpvect =: ((#txt_n),(#dict_words)) $ (1{,./ >> each
corpus)
1 1 0 0 0 0
0 0 1 1 0 0
1 0 1 1 0 0
0 0 2 1 1 1

NB. Extract search bow vector
]searvect =: >1{search_bow
1 1 0 0 0 0
```

Step 9:

Calculate the inverse document frequency (idf) of each word/term. The idf is calculated by determining in how many documents (txt_0-3) does word_n appear?

Question: In how many documents does word_0 (sky) appear?
Answer: word_0 (sky) appears in 2 documents (txt_0 & txt_2).

Doing this for all words/terms, results in:

```
NB. Calculate inverse document frequency for each word in
NB. dictionary
]idfs =: > get_idf each dict_words
2 1 3 3 1 1
```

Step 10:

Calculate the idf weight for each word/term using the formula:

$$w = log_2 \frac{n}{f}$$

w = weight
n = number of documents
f = inverse document frequency

Doing this for all words/terms in idfs, results in:

```
NB. Calculate the idf weight
]idfs =: 2^.(#txt_n)%idfs
1 2 0.415037 0.415037 2 2
```

Step 11:

Multiply each term frequency in the search bow with the inverse document frequency weight calculated in step 10.

$(0,1) \Rightarrow 1 \times idfs[0] = 1 \times 1.0 = 1.0 \Rightarrow (0,1.0)$
$(1,1) \Rightarrow 1 \times idfs[1] = 1 \times 2.0 = 2.0 \Rightarrow (1,2.0)$

Then normalize the resulting vector using the formula:

$$n_1 = \frac{w_1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_n^2}}$$

Example:

$$n_0 = \frac{1.0}{\sqrt{1.0^2 + 2.0^2}} = 0.4472135954999579$$

$$n_1 = \frac{2.0}{\sqrt{1.0^2 + 2.0^2}} = 0.8944271909999159$$

J Code to do all these calculations:

```
NB. Calculate tfidf vector
]tfidf_vect =: normalize (searvect * idfs)
0.447214 0.894427 0 0 0 0
```

Step 12:

Calculate the corpus idfs.

```
NB. Calculate corpus idfs
]corpus_idfs =: corpvect *"1 idfs
1 2        0        0 0 0
0 0 0.415037 0.415037 0 0
1 0 0.415037 0.415037 0 0
0 0 0.830075 0.415037 2 2
```

Step 13:

Normalize all vectors in corpus_idfs using the formula from step 11.

$$n_1 = \frac{w_1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_n^2}}$$

```
NB. Calculate normalized corpus idfs
]norm_idfs =: normalize"1 corpus_idfs
0.447214 0.894427        0        0        0        0
       0        0 0.707107 0.707107        0        0
0.862418        0 0.357936 0.357936        0        0
       0        0 0.278849 0.139424 0.671865 0.671865
```

Step 14:

Calculate the cosine similarity (score value), which is the dot product of each vector (row) in corpus_idfs and tfidf_vect, the query/search vector.

```
NB. Calculate cosine similarity
]cosine_similarity =: +/ |: norm_idfs *"1 tfidf_vect
1 0 0.385685 0
```

Step 15:

Finally we create a nice and human readable output from the resulting vector of step 14.

```
NB. Create nice output
]docsim =: ('Document';'Score'),((<"0
i.#cosine_similarity),.(<"0 cosine_similarity))
```

Document	Score
0	1
1	0
2	0.385685
3	0

That means, the query "The sky is blue" matches the documents (txt_0-3) as follows:

Document	Score	Match	Original Text
0	1	100%	The sky is blue
1	0	0%	The sun is bright
2	0.385685	38%	The sun in the sky is bright
3	0	0%	We can see the shining sun, the bright sun

Real World Example

As mentioned earlier, here is the real world example, which performs much better than the educational example:

```
NB. Setup: Files to read
file_dir =: '~user/Example_Data6'
dat_fils =: 0{ |: 1!:0 (jpath file_dir,'/*.dat')

NB. Setup: Example search
search_list =: 'ethical';'business';'practices';'reputation'

NB. Util: Read all dat-files
read_all_files =: 3 : 'LF cut (fread (file_dir,''/'',y))'

NB. Util: Dictionary frequency
dict_freq =: 3 : '<: #/.~ (dict_words,y)'

NB. Util: Normalize a vector
normalize =: 3 : 'y%(%:+/y^2)'

NB. Util: Sort table x=column y=table
sort_desc =: ]\:{"1

NB. Analyze: Read all files
txt_n =: read_all_files each dat_fils

NB. Analyze: Create dictionary (unique id for each word)
dict_words =: (~.(,/>txt_n))-.<''

NB. Analyze: Create corpus
corpus =: > dict_freq each txt_n

NB. Analyze: Create search bow
search_bow =: $. (i.#dict_words){ dict_freq search_list

NB. Analyze: Calculate inverse document frequency
idfs =: $. 2^.(#txt_n)%(+/ 0 >~ corpus)

NB. Analyze: Calculate tfidf vector
tfidf_vect =: $. normalize (search_bow * idfs)

NB. Analyze: Calculate corpus idfs
corpus_idfs =: $. corpus *"1 idfs

NB. Analyze: Calculate normalized corpus idfs
norm_idfs =: $. normalize”1 corpus_idfs

NB. Analyze: Calculate cosine similarity
cosine_similarity =: $.^:_1 +/ |: norm_idfs *"1 tfidf_vect

NB. Analyze: Create document similarity array
docsim =: dat_fils,.(<"0 cosine_similarity)

NB. Output: Create nice output sorted descending on score
docsim_sorted =: ('Document';'Score'),(1 sort_desc docsim)
```

The folder "~user/Example_Data6" (as used in the script) contains pre-processed .dat files. The pre-processing includes the following tasks:

- Extraction of text from source PDF files
- Converting extracted text to lowercase
- Elimination of punctuation characters
- Elimination of stop-words
- The .dat files contain "one word per line"

Please note:

In contrast to Python/gensim, the J program does NOT use any library functions except:

- 1!:0 Read the contents of a directory
- fread Read the contents of a file (just a user friendly implementation of 1!:1)
- cut Cut text on a character (LF / linefeed in this example)
  ```
  cut =: ' '&$: :([: -.&a: <;._2@,~)
  ```
- each Execute a verb (function) on each cell of an array
  ```
  each =: &.>
  ```

All the other statements are pure core elements of the J programming language and its built-in array handling capabilities.

Number of program code lines J / Python (gensim) (both w/o comments)

Python uses various libraries like gensim which rely on numpy, scipy, and many other packages. The total number of lines of Python program code are estimated by debugging through the code, and by diving into the various library functions.

	J	Python / gensim
Read all files	5	~20
Create dictionary	1	1 (gensim: >100)
Create corpus	2	1 (gensim: >100)
TOTAL preparation code lines	**8**	**~22 (gensim: >200)**
Create search bow	2	1 (gensim: >100)
Calculate idf dictionary	1	1 (gensim: >100)
Calculate tfidf vector	2	1 (gensim: >100)
Calculate corpus idfs	1	1 (gensim: >100)
Normalize corpus idfs	1	1 (gensim: >100)
Calculate cosine similarity	1	1 (gensim/numpy/scipy: >300)
Sort output	2	1
TOTAL calculation code lines	**10**	**7 (gensim/numpy/scipy: >800)**
TOTAL	**18**	**~29 (gensim/numpy/scipy: >1000)**

Performance comparison J / Python (gensim)

This "Real Word Example" was tested with 160 .dat files containing more than 2'800'000 words. The dictionary size was more than 45000 different words. The computer was a Mac with a 2.13 GHz Intel Core 2 Duo CPU with 4GB of RAM, and an SSD disk.

The following performance comparison shows where J is faster, and where it's slower, compared to the Python/gensim solution. All values are in seconds.

	J	Python / gensim
Read all files	1.86	1.11
Create dictionary	2.10	2.93
Create corpus	2.96	2.36
TOTAL preparation time	**6.92**	**6.40**
Create search bow	0.04	0.01
Calculate idf dictionary	0.03	0.25
Calculate tfidf vector	0.01	0.01
Calculate corpus idfs	0.08	0.01
Normalize corpus idfs	0.30	0.01
Calculate cosine similarity	0.16	2.21
TOTAL calculation time	**0.62**	**2.50**
TOTAL	**7.54**	**8.90**

What about R?

R [8] is widely used in statistics as well as in data analysis and data mining. R is known as the so called "lingua franca" of statistics. But is it a good solution for that kind of problems?

Yes and no. Some built-in array handling capabilities in R makes the code elegant, and may reduce the number of code lines, if you avoid loops where it's possible. The speed of R is mostly equivalent, or somewhat slower than Python (at least in my experiments). R was clearly built to make the life of the statistician easier (that's matter of opinion), not the life of the computer.

Conclusion

The overall performance of J is about 15% better than the Python/gensim solution. But I'm sure that J's preparation part can be even more optimized. Any volunteers?

The calculation speed of the Tf-Idf vectors and the cosine similarity is about four times (400%) faster in J, compared to Python/gensim. This is where J can unlock its full power. So if you are going to use the same set of documents (corpus) for different queries: J is your friend.

References

[1] The Python Software Foundation
https://www.python.org

[2] Radim Rehurek - gensim / topic modelling for humans
https://radimrehurek.com/gensim/index.html

[3] Jsoftware - High-performance development platform
http://www.jsoftware.com

[4] Christian S. Perone - Machine Learning :: Text feature extraction (tf-idf) Part I
http://blog.christianperone.com/?p=1589

[5] Christian S. Perone - Machine Learning :: Text feature extraction (tf-idf) Part II
http://blog.christianperone.com/?p=1747

[6] Christian S. Perone - Machine Learning :: Cosine Similarity for Vector Space Models Part III
http://blog.christianperone.com/?p=2497

[7] Cambridge University Press - The vector space model for scoring
https://nlp.stanford.edu/.../the-vector-space-model-for-scoring-1.html

[8] The R Project for Statistical Computing
https://www.r-project.org

Tboxes and Boolean Normal Forms[1]

Nollaig MacKenzie
3781 Panorama Crescent
Chemainus, BC V0R 1K4
CANADA

1 The functions used herein can be obtained from:

ftp://ftp.nollaig.ca/pub/pap/tbox

Preamble: Tboxes

A tbox[2], in my sense, is essentially an ordered pair whose items are a string of N letters and an N#2 boolean array[3]. In J, just box the two items.

Tboxes can stand in for classes of equivalent boolean schemata: ('ab';2 2$ 0 1 1 0) would give the content of 'a~:b', 'a=-.b', and so on.

To apply the boolean functions to tboxes, I use the adverb *tbfn*:

```
1 : 0
'xr yr'=. y
xr unpad u yr
:
'xl yl xr yr'=. x,y
N=. #ov=. xl ([ #~ [ e. ]) xr
ll=. (xl-.ov),(xr-.ov),ov
I=. i.&ov@[|:]
ll unpad (xl I yl) (u"N"(N,_)) xr I yr
)
```

Other adverbs and conjunctions will be defined, but *tbfn*[4] is central.

An example using *tbfn*:

```
   ('ab';2 2$1 0 0 1) +. tbfn 'b';0 1
```

ab	1 1
	0 1

('(a=b)+.b' has the same truth-table as 'a<:b').

Normal Forms:

Abstractly, a normal form for a schema S is a schema Sn with the same truth-table as S, where Sn satisfies the syntactical conditions:

(1) The only truth-functions occurring in Sn are the

---

2 See my essay in *Journal of J* v4 n1. “tbox” may have been an unfortunate choice of term; it is used elsewhere in different senses. But I am sure there will not be confusion.

3 The first item of a tbox need not be a string of letters. It merely provides something like indices to the axes of the second item, so integers would do, or, indeed, anything to which /: would apply.

4 Tbfn will work, of course, for functions and arguments other than boolean. Try: ('abc';i.2 3 4) j. tbfn 'bcd';i.3 4 5.

superordinate function F, the subordinate function f, and negation (-.).

(2) No F nor f occurs in the scope of a negation.

(3) No F occurs in the scope of an f.

Where F is +. and f is *., Sn is an *alternational* normal form (ANF); where F is *. and f is +., a *conjunctional* normal form (CNF).

For example, suppose EX01 is

'a<:(b=c)*.a'.

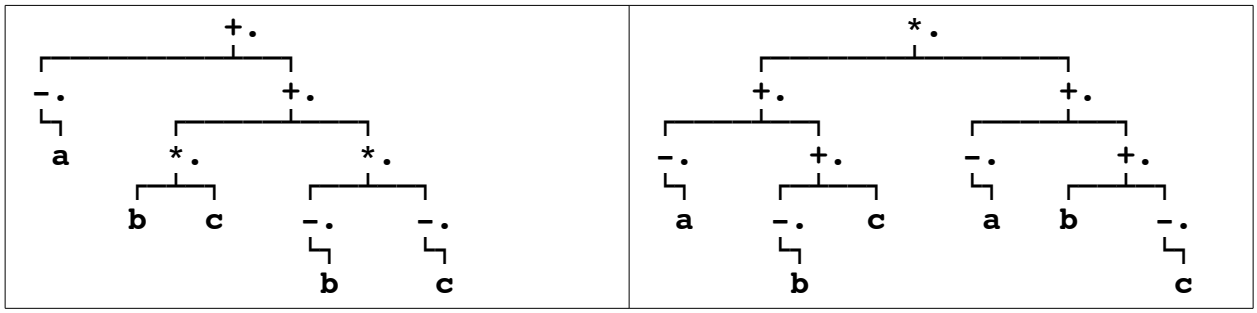
An ANF for EX01 might be EX01anf

'(-.a)+.(b*.c)+.(-.b)*.-.c',

and a CNF might be EX01cnf

'((-.a)+.(-.b)+.c)*.(-.a)+.b+.-.c'.

Their satisfaction of the syntactical requirements can be seen from their trees:



Standardly, normal forms are produced by repeated "algebraic" substitutions:

(1) Replace all occurrences of unwanted function signs with equivalences using '-.', '+.', and '*.', as in:[5]

⌜S1=S2⌝ => ⌜(S1*.S2)+.(-.S1)*.-.S2⌝

(2) Push negations across '+.' and '*.' using DeMorgan's Rules; e.g.:

⌜-.S1+.S2⌝ => ⌜(-.S1)*.-.S2⌝

5 ⌜ and ⌝ are Quine's "corner quotes". They allow one to mix names of expressions with actual strings. ⌜-.S1+.S2⌝ should be taken as: '-.','(',S1,')','+.','(',S2,')'. Quine, Selected Logic Papers, p. 101ff.

(3) If an F is in the scope of an f, rewrite; e.g.:

⌜S1*.S2+.S3⌝ => ⌜(S1*.S2)+.S1*.S3⌝

A normal form will ordinarily be a number of *clauses*, linked by F; in Sanf, '-.a', 'b*.c', '(-.b)*.-.c', where each clause is a number of *literals* (letters, either negated or unnegated), linked by f.

Normal forms by tbox

For an ANF, the truth-table for each clause will be all 0s but for a single 1.[6] So the tbox version of an ANF will be a stack of tboxes, each with a single 1 in the second element. For EX01 and EX01anf, the tbox versions will be:

EX01tb

abc	1 1
	1 1
	1 0
	0 1

EX01tbanf

a	1 0
bc	0 0
	0 1
bc	1 0
	0 0

EX01tb is implied by each clause of EX01tbanf, and

EX01tb -: tbal +. tbfn/ EX01tbanf[7]

I approach the discovery of clauses not through algebraic manipulation of schemata, but by actions on tboxes.

---

6 What one says about an ANF is mirrored in what one says about a CNF, if one interchanges: 0 and 1; boolean functions and their duals; “implies” and “is implied by”. The function duals are:

```
*. >  <  +: =
+. >: <: *: ~:
```

The dual of a truth-table is given by: dual=: $ $ -.@:|.@,

If ***A*** is an ANF for T, and ***C*** is a CNF for the dual of T, the items of ***C*** will be duals of those of ***A***.

7 tbal=: **/:L:0@:{. (([{L:0 {.@:]) , [|:L:0 {:@:])]**
simply puts the letters and axes of a tbox in /: order.

Consider *tbredax*:

```
1 : 0
:
'll rr'=. y
tunpad (ll -. x);u/"(ll (#@[ - i.) x) rr
)
```

It will perform a *u* reduction on the axis corresponding to the letter in x :

```
   EX02
┌───┬─────┐
│abc│1 0  │
│   │1 1  │
│   │     │
│   │0 0  │
│   │0 1  │
└───┴─────┘
```

```
   ]EX02lis=: ('a' *. tbredax EX02) ,: 'b' *. tbredax EX02
┌──┬─────┐
│bc│0 0  │
│  │0 1  │
├──┼─────┤
│ac│1 0  │
│  │0 0  │
└──┴─────┘
```

It is evident that if R=: A *. tbredax B, R must imply B:

Imagine that Rp is R padded with letter A back to the shape of B. The padding may put a 0 in a cell where B has a 1, but never a 1 where B has a 0. So in no cell will ⌜Rp <: tbfn B⌝ have a 0. But Rp is just two copies of R catenated on the padding axis; so neither will ⌜R <: tbfn B⌝ have a 0 in any cell.

Illustration:

```
    (tbal 'a' tbpad 0{EX02lis) ,&< EX02
┌──────────┬──────────┐
│┌──┬─────┐│┌───┬─────┐│
│|abc|0 0  |||abc|1 0  ||
│|  |0 1  |||   |1 1  ||
│|  |     |||   |     ||
│|  |0 0  |||   |0 0  ||
│|  |0 1  |||   |0 1  ||
│└──┴─────┘│└───┴─────┘│
└──────────┴──────────┘
```

The function *tbfcl* applies tbredax to each axis of its right argument, as deeply as possible:

```
1 : 0
(0 2$'') u tbfcl y
:
val=. u/i.0
ll=. >@{. y
for_il. i. # ll
do.
r=. (il{ll) u tbredax y
  if. val e. , > {: r do.
  s=. u tbfcl r
    if. 0 2 +. . ~: $s do.
    x=. x,s
    else.
    x=. x,r
    end.
  end.
end.
~. x
)
```

For example,

```
   *. tbfcl 'abc';truarr 'a<:b*.c'
┌──┬────┐
│bc│0 0 │
│  │0 1 │
├──┼────┤
│a │1 0 │
└──┴────┘
```

Applying tbfcl to EX02, we get:

```
   ]EX02lis2=: *. tbfcl EX02
┌──┬────┐
│bc│0 0 │
│  │0 1 │
├──┼────┤
│ac│1 0 │
│  │0 0 │
├──┼────┤
│ab│0 1 │
│  │0 0 │
└──┴────┘
```

EX02 -: +. tbfn/ EX02lis2, so EX02lis2 is an ANF for EX02.

⌜*. tbfcl A⌝ gives correct results, in the sense that it yields a list (possibly empty) of tboxes each of which implies A. But it may fail, in two ways:

(1) The list may be incomplete: ⌜A -: +. tbredu *. tbfcl A⌝ may be false.

(2) The second element of a tbox in ⌜*. tbfcl A⌝ may contain more than one 1.

(Failure 1): Consider the tbox EX03:

```
┌────┬────┐
│pqrs│1 1 │
│    │1 1 │
│    │    │
│    │1 0 │
│    │1 0 │
│    │    │
│    │    │
│    │0 1 │
│    │0 0 │
│    │    │
│    │1 1 │
│    │1 0 │
└────┴────┘
```

If we apply tbfcl, we get

```
] EX03fc=: *. tbfcl EX03
┌──┬────┐
│qs│0 0 │
│  │1 0 │
├──┼────┤
│ps│1 0 │
│  │0 0 │
├──┼────┤
│pq│1 0 │
│  │0 0 │
└──┴────┘
```

But EX03fc is not an ANF for EX03. Take

```
   ] EX03fcsum=: tbal +. tbfn/ EX03fc
┌───┬────┐
│pqs│1 1 │
│   │1 0 │
│   │    │
│   │0 0 │
│   │1 0 │
└───┴────┘
```

If we padded EX03fcsum by 'r', we would find a deficit of two 1s.

We can find the “extra” item by noting that:

If A implies B, then B is equivalent to ⌜A+.(-.A)*.B⌝.

If A implies B, there will be no cells where A has a 1 and B has a 0. ⌜(-.A)*.B⌝ will have a 1 at each cell where B has a 1 and A has a 0. Therefore ⌜A+.(-.A)*.B⌝ will have 1s at exactly the cells B does.

So (with EX03fcsum for A, EX03 for B),

```
   ]extra=: tbal EX03fcsum < tbfn EX03
┌───┬─────┐
│prs│0 0  │
│   │0 0  │
│   │     │
│   │0 1  │
│   │0 0  │
└───┴─────┘
```

Let EX03fce=: EX03fc,extra. Then

```
 EX03 -: tbal +. tbfn/ EX03fce
```

EX03fce is thus a tbox ANF for EX03.[8]

```
┌───┬─────┐
│qs │0 0  │
│   │1 0  │
├───┼─────┤
│ps │1 0  │
│   │0 0  │
├───┼─────┤
│pq │1 0  │
│   │0 0  │
├───┼─────┤
│prs│0 0  │
│   │0 0  │
│   │     │
│   │0 1  │
│   │0 0  │
└───┴─────┘
```

8 As it happens, one of its clauses is redundant.

(Failure 2): Here is a nice example:

EX04

pqr	0	1
	1	1
	1	1
	1	0

Apply tbfcl:

]EX04a=: *. tbfcl EX04

qr	0	1
	1	0
pr	0	1
	1	0
pq	0	1
	1	0

In fact, any one of the items in EX04a can be dropped without loss. Arbitrarily, suppose we make EX04a

qr	0	1
	1	0
pq	0	1
	1	0

EX04 -: tbal +. tbfn/ EX04a, indeed, but the 1s on the diagonals make EX04a not an ANF. The solution is straightforward; expand EX04a thus:

EX04anf

qr	0	1
	0	0
qr	0	0
	1	0
pq	0	1
	0	0
pq	0	0
	1	0

These devices correct the inadequacies of tbfcl. But with regard to (2) it might be worth taking up a more relaxed idea of normal forms. After all, if EX04a and EX04anf are translated into schemata,

```
EX04a:   (q~:r)+.p~:q
EX04anf: (q<r)+.(q>r)+.(p<q)+.p>q
```

EX04a's is easier to understand.

Simplification

Let us turn now to techniques for *simplifying* normal forms. The concept is straightforward: If removal of a clause, or an axis in a clause, leaves unaltered the pattern of 1s and 0s under F-reduction, the clause or axis is redundant.

For *redundancy of a clause*, Quine's test, recast to fit tboxes, is:

> **ANF: the clause implies the +./ of the rest of the ANF.**
>
> **CNF: the clause is implied by the *./ of the rest.**

The function that does the test is tbcredun:

```
   tbcredun
2 : (':'; '(x{y) u luvredr v (y -. x{y)')
   luvredr
2 : (':'; 'x u tbfn (v tbredu y)')
   tbredu
1 : 'if. (1 = #@$ y) +. 0 e. $y do. y else. u tbfn/ y end.'
```

If the subject is an ANF, u will be <:, v will be +., and a tautology indicates a positive result. If a CNF, u will be <, v will be *., and a contradiction indicates a positive result.

Imagine EX04a again as a 3 item array, *. tbfcl EX04:

```
┌──┬────┐
│qr│0 1 │
│  │1 0 │
├──┼────┤
│pr│0 1 │
│  │1 0 │
├──┼────┤
│pq│0 1 │
│  │1 0 │
└──┴────┘
```

```
   1 <: tbcredun +. EX04a
┌─┐
│1│
└─┘
```

That ⌜A <: B⌝ is a tautology means, of course, that A implies B. Notice that the number of 1s in the clauses is irrelevant to the test — it is not required that EX04a be an ANF, strictly.

Clearly the test is sound: Suppose B is a stack of tboxes, and B* is B without item I. Call the +.-reduction on B, OB, that on B*, OB*. The 1-cells of OB are just the 1-cells of OB* plus whatever extra 1-cells are provided by I. But if I implies OB*, there is no cell where I has a 1 and OB* has a 0. So OB* and OB match completely, and B* may replace B.

The test for *redundancy of an axis of a clause*:

ANF: The clause without that axis implies the +.-reduction of the whole ANF.

CNF: The clause without that axis is implied by the *.-reduction of the whole CNF.

For the test to be sound, the tbox need not strictly fit the definition of “clause of an ANF”. Call the second part of the tbox — the boolean array — C; we needn't require that C have only one 1; what we need is: if C is transposed to bring the relevant axis to the front, then either {.C or {:C is all 0s. This corresponds to saying, if L is the letter in question, that a related schema can be rephrased as 'L*.S' or '(-.L)*.S'.

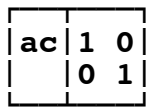
If {.C on the chosen axis is all 0s, the second part of the tbox without the axis is {:C, and vice-versa; the first part is just the letters of the original without L; thus the new tbox is just:

```
    L +. tbredax the_old_tbox
```

For example, if our clause were EX05o:

```
   (,&< 0 2&tbtran) EX05o
┌─────────┬─────────┐
│┌───┬───┐│┌───┬───┐│
│|abc|0 0|||bac|0 0||
│|   |1 0|||   |0 0||
│|   |   |||   |   ||
│|   |0 0|||   |1 0||
│|   |0 1|||   |0 1||
│└───┴───┘│└───┴───┘│
└─────────┴─────────┘
```

EX05n=: 'b' +. tbredax EX05o, would be:



Suppose, then, a list of clauses, *OLD* (C0, …, Cn), whose +. tbfn reduction is Csum, and we find that D, which is Ci without the axis corresponding to letter *1*, implies Csum. Given that implication, D can be added to (C0, …, Cn) to form an equivalent list, *NEW* (C0, …, Ci, D, …, Cn).

Ci must imply D, since they are related exactly as Ex05o and EX05n are. That is, when ⌜Ci <: tbfn D ⌝ is evaluated, the pairing that tbfn does will yield either ⌜(($D)$0) <: D⌝ or ⌜D <: D⌝.

If Ci implies D, it implies the rest of *NEW* (that is, *NEW* without Ci), and is therefore redundant. *NEW* without Ci may therefore replace *OLD*, or, to shorten the story, in *OLD* D may replace Ci.

All this can be put together in a function *tbnorm*. It produces results at least as good as functions that work by manipulation of character strings, and is anywhere from 5 to 4000 times faster, on examples that I have tried. Perhaps not much can be inferred from that; I wrote the character string functions without much thought about efficiency. But the basic shape of a function to produce a normal form by working on character strings will be: get the syntactic structure of a schema, rewrite, repeat. This takes a lot of computational labour. Working on tboxes is bound to be quicker.

Tbnorm produces a normal form in the strict sense; each clause contains just one occurrence of its specific value. That, after all, was the point of the exercise. But the resources developed here could serve other aims.

One can hope that someimes a normal form for a schema will exhibit its content in a more perspicuous way. But by no means always; the normal forms for 'p*:s*.q=r':

```
((-.p)+.q+.r+.-.s)*.(-.p)+.(-.q)+.(-.r)+.-.s
```

```
(-.p)+.(-.s)+.((-.q)*.r)+.q*.-.r
```

would be no easier to grasp than the original. 'p<:s<:q~:r', however, which a slightly truncated *tbnorm* might give us, is the most perspicuous of all. In the background of this essay see a not very clearly defined project to write a J function that will tease out of a tbox that most perspicuous schema.

Or perhaps: Think of 0s and 1s scattered in an N-dimensional cube, forming constellations which can inhabit cubes of lesser rank; by building the greater out of the lesser we better reveal its structure.

References:

Quine, W V, Methods of Logic, 4th ed., Harvard University Press, chs. 10-11.

Quine, W V, Selected Logic Papers, Random House, 1966, ch. XIV. (In this essay I have not explored tbox versions of cores and prime implicants.)

Pick, Points, and Polygons expressed in J

John C McInturff

(6-2-2017)

Abstract

This article is an examination of some of the basic properties of polygons in the plane, including that of Georg Pick. Approximate values for some of these results are easily obtained using a ruler and a protractor. However, more precise results are required in many fields, as in navigation, surveying, finance, etc., and are worthy of using the best mathematical practices. This article illustrates the use of J and complex vectors.

Background

Consider P a finite set of points in the X-Y plane. For illustrative purposes, and without loss in generality, consider 5 points having the x,y coordinate shown in Figure 1.

```
   points=. 1 2 3 4 5
   x=. 4 7 8 6 3
   y=. 2 3 6 7 5

   x;'';y
```

4 7 8 6 3	2 3 6 7 5

```
   (,.points);'';(P=. x,.y)
```

1	4 2
2	7 3
3	8 6
4	6 7
5	3 5

Connecting each point in sequence by a line forms a closed polygon where the points are the polygon's vertices. (Figure 2)

The resulting polygon, is introduced early in mathematical classrooms[1] and is revisited many times in subsequent years to illustrate a particular mathematical concept, 14 of which are illustrated here.

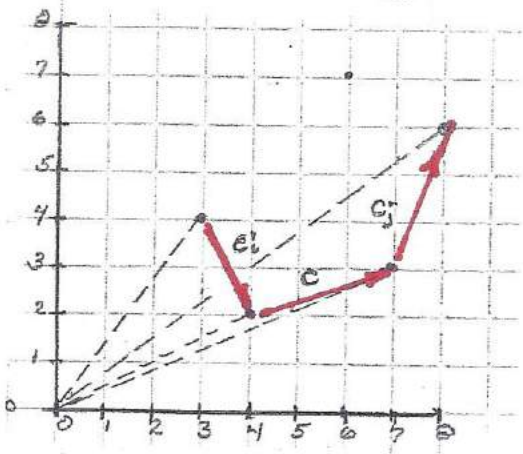


Figure 1

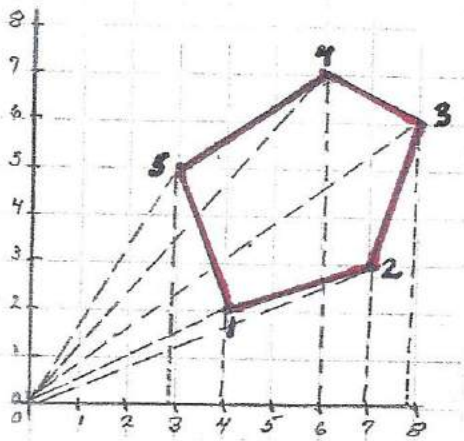


Figure 2

Approach

Since the set of points of a polygon are fixed in two-dimensional space, each point can be represented by a complex number. In this example, an array of the complex numbers representing each vertex therefore describes the polygon. Since the elements of the vector thus formed are complex numbers, the resultant vector is a complex vector. It is reasonably well known that complex vectors find use in electric circuit and electromagnetic field theory, and in transmission line analyses. However, in these applications, the effective phase angles of the complex vectors are a function of additional parameters, such as time and distance.

Analysis

1a Polygon representation

```
   ]p=. x j. y
4j2 7j3 8j6 6j7 3j5

   (,.points);'';P;'';(,.p)
```

1	4 2	4j2
2	7 3	7j3
3	8 6	8j6
4	6 7	6j7
5	3 5	3j5

1b Polygon components, c

```
   '`r rr'=. (1|.])`(_1|.])
```

In this approach, the components of the polygon are expressed as the complex vector, c, where:

```
   c=. (r-]) p
   Show c
```

3j1	1j3	_2j1	_3j_2	1j_3

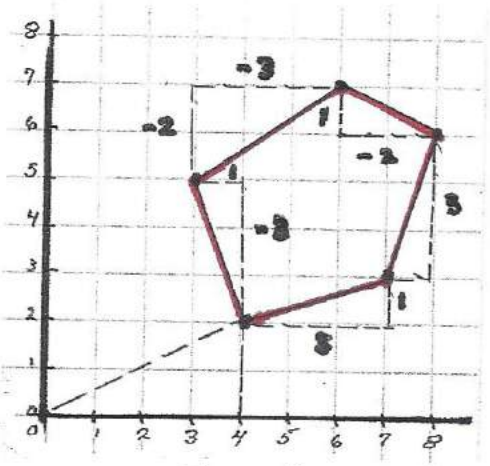


Figure 3

2. Distance, d, between each point

The distances sought is the magnitude of c.

```
   Show d=. |c
```

3.162	3.162	2.236	3.606	3.162

3. Perimeter of polygon

Summing each value of d produces the perimeter of the polygon.

```
   ]Perimeter=. +/d
15.33
```

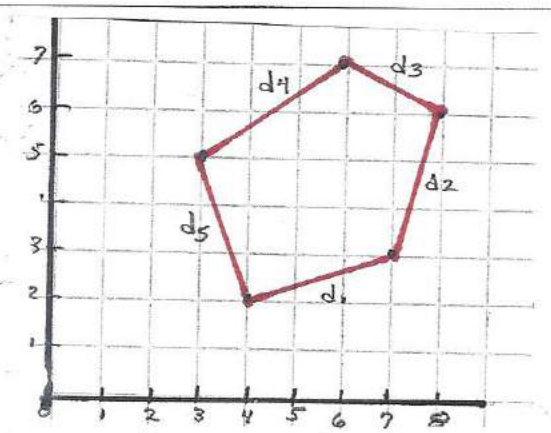


Figure 4

4. Exerior angles, alpha

See Figures 1,3,5

```
   Show 'ci cj'=. (r,:rr) c
```

1j3	_2j1	_3j_2	1j_3	3j1
1j_3	3j1	1j3	_2j1	_3j_2

Figure 1 depicts ci,c,cj.
The dyad, o=. *+, is the complex vector product.

```
   Show v=. c o cj
```

0j10	6j8	1j7	4j7	3j11

The polar form of the complex vector, v has a magnitude, V=. |v, and phase angle, alpha. The items of the phase angle, alpha are the exterior angles, sought.

```
   Show alpha=. deg v
```

90	53.13	81.87	60.26	74.74

Note that these angles sum to 360 degrees, as they should.

```
   +/alpha
360
```

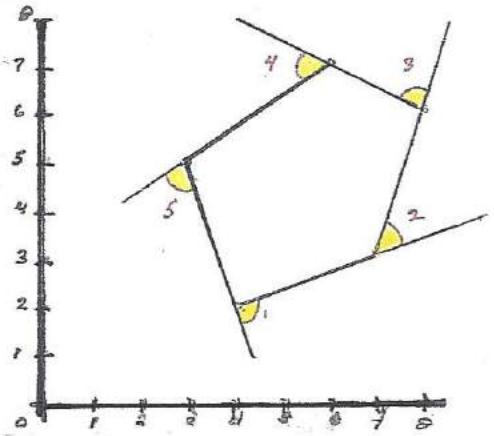


Figure 5

5. Interior angles, beta

```
   Show beta=. 180-alpha
```

90	126.9	98.13	119.7	105.3

Note that these sum to 540, as they should.

```
   +/beta
540
```

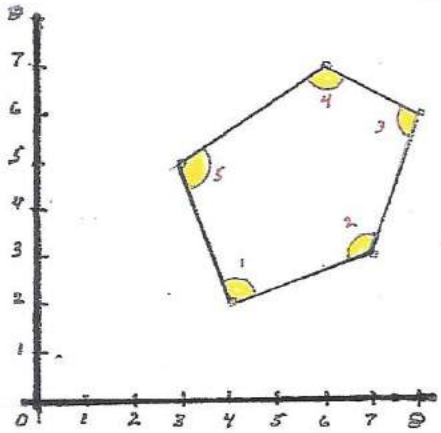


Figure 6

Area

The area, A, of the polygon is produced by the monad, F, the basis of which comes from Heron of Alexandria, also called Hero (c. AD 62), and incorporated into APL/J by K. Iverson.

```
   F=. 13 : '-: +/ 2 det\ +.(,0{[]) y'
   ]A=. F p
15.5
```

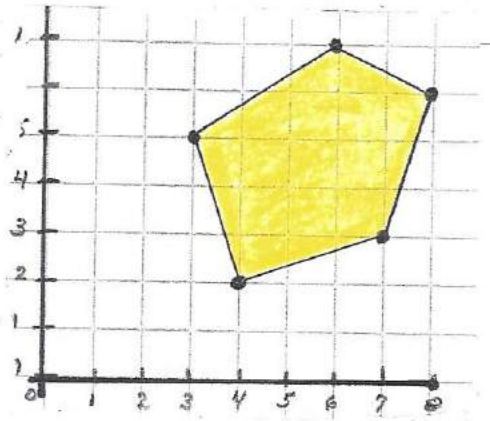


Figure 7

7. Pick's Area

Integer coordinates are called lattice points. On a lattice polygon, every vertex and lattice points in between vertices are called exterior lattice points. The interior, integer, coordinate are called *interior* lattice points. It is seen from Figure 8 that:

```
   x=. 14 interior lattice points,
   y=. 5 exterior lattice points
```

An elegant, useful, and simple relationship between the *lattice points* and the area of a polygon was published by Georg Alexander Pick. (See Reference)

Pick's relationship is expressed here as the dyadic hook

```
    PICK=. + [:<:-:
    ]Area=. 14 PICK 5
15.5
```

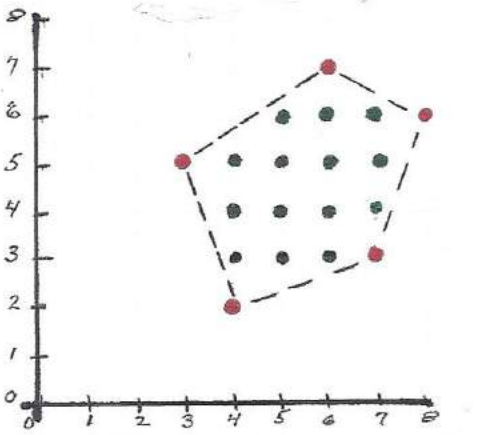


Figure 8

8. Primitive Area

The Area, of this polygon is easily, but not always practically, obtained by subtracting the area of the shaded bordering triangles and rectangles, Bt, from the total area of the polygon's bordering frame, Bf, For this particular polygon

```
   ]Bt=. >:-: ip/|:k=.|+.C
9.5

   ]Bf=.  */ -: +/k
25

   ]Area=. Bf - Bt
15.5
```

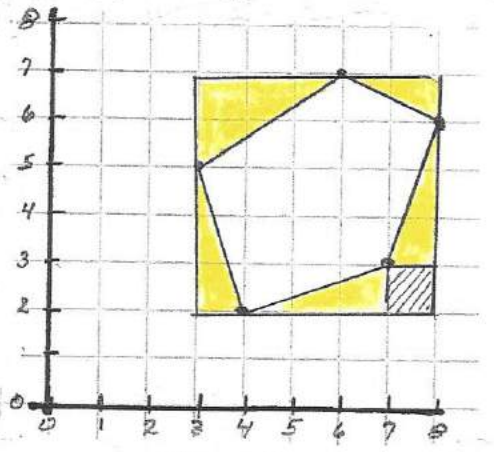


Figure 9

9. Rotate

```
rotate=.cjs 30 0 _30
Show rotate */p
```

2.5j3.7	4.6j6.1	3.9j9.2	1.7j9.1	0.0j5.8
4j2	7j3	8j6	6j7	3j5
4.5j_0.27	7.6j_0.9	9.9j1.2	8.7j3.1	5.1j2.8

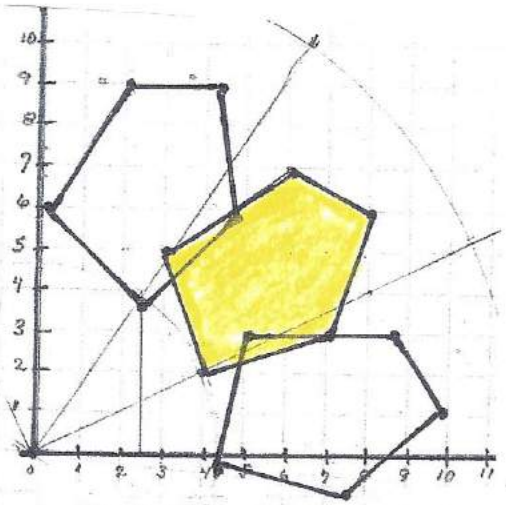


Figure 10

10. Scale (Expansion / Contraction)

```
S=. 0.5 1 1.5
Show S */ p
```

2j1	3.5j1.5	4j3	3j3.5	1.5j2.5
4j2	7j3	8j6	6j7	3j5
6j3	10.5j4.5	12j9	9j10.5	4.5j7.5

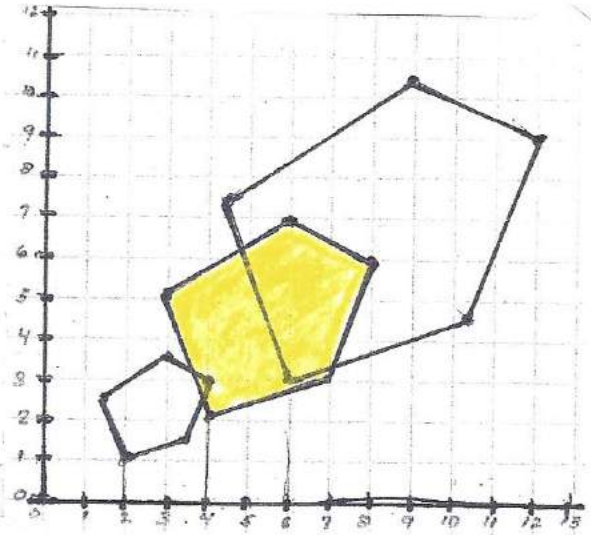


Figure 11

11 Translation

```
T=. 8j7 0 10j2
Show Translate=. T +/ p
```

12j9	15j10	16j13	14j14	11j12
4j2	7j3	8j6	6j7	3j5
14j4	17j5	18j8	16j9	13j7

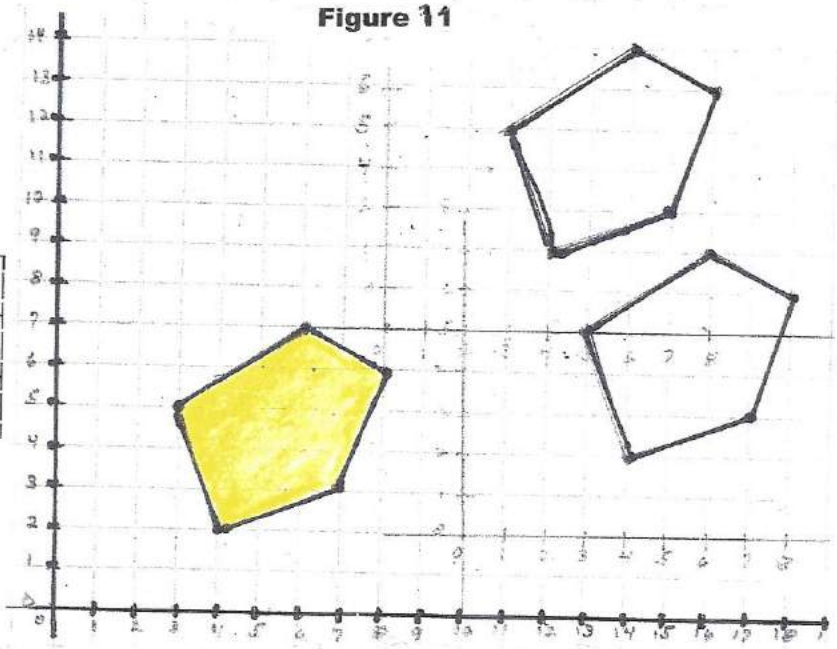


Figure 12

Pick, Points, and Polygons expressed in J

John C McInturff

Background

Consider P a finite set of points in the X-Y plane. For illustrative purposes, and without loss in generality, consider 5 points having the x,y coordinate shown in Figure 1.

```
points=. 1 2 3 4 5
x=. 4 7 8 6 3
y=. 2 3 6 7 5

x;'';y
```

4 7 8 6 3	2 3 6 7 5

```
(,.points);'';(P=. x,.y)
```

1	4 2
2	7 3
3	8 6
4	6 7
5	3 5

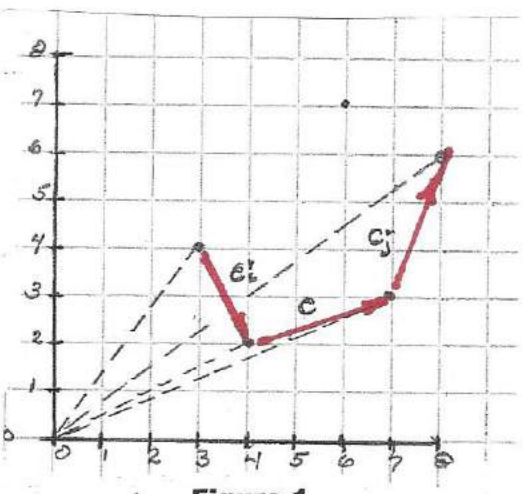


Figure 1

Connecting each point in sequence by a line forms a closed polygon where the points are the polygon's vertices. (Figure 2)

The resulting polygon, is introduced early in mathematical classrooms[1] and is revisited many times in subsequent years to illustrate a particular mathematical concept, 14 of which are illustrated here.

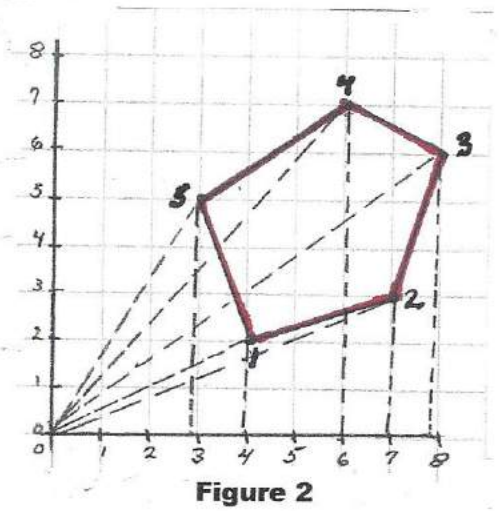


Figure 2

Approximate values for some of the results to be shown are easily obtained using a simple ruler and protractor. However, when precise values are required (as in navigation, surveying, finance, etc.), considerable effort could result unless the best mathematical practices are applied.

12. Distort

```
   Show Do=. (0.5 1.5);(1 1);(1.5
0.5)
```

0.5 1.5	1 1	1.5 0.5

```
   (>Do);'';(|;+.p)
```

0.5 1.5 1 1 1.5 0.5	4 2 7 3 8 6 6 7 3 5

```
   fo=. ,:/ . *

   ]D=. (>Do) fo (|:+. p)
2  3.5  4    3 1.5
3  4.5  9 10.5 7.5

4    7  8    6   3
2    3  6    7   5

6 10.5 12    9 4.5
1  1.5  3  3.5 2.5
```

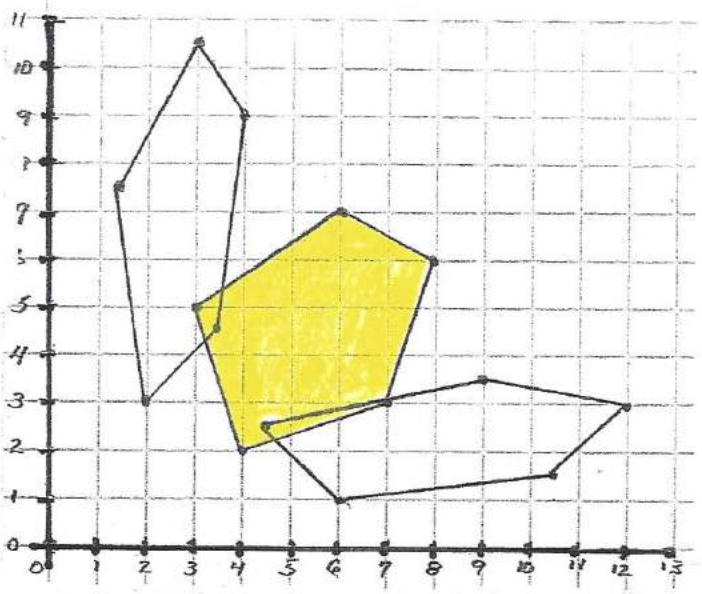


Figure 13

```
Show Distort=. j./"2 D
```

2j3	3.5j4.5	4j9	3j10.5	1.5j7.5
4j2	7j3	8j6	6j7	3j5
6j1	10.5j1.5	12j3	9j3.5	4.5j2.5

14 Resultant force, distance

Vertical forces, f, placed on the polygon at the points shown below are equivalent to a single force, F placed at the point D.

```
   Show points,:f=. 3 1 _2 6 4
```

1	2	3	4	5	6
3	1	_2	6	4	0

In other words:

```
Requirement=. '(F*D)-:(f ip p)'
   ]F=. +/f
12
   (f ip p)
51j59

Solve for unknown, D ;using dyad G
   G=. ip % [:+/[
   D=. f G p
   Show D ;(+.D)
```

4.25j4.917	4.25 4.917

The real and imaginary parts of D are the x and y coordinates of D,

```
Is the requirement satisfied?
   ". Requirement
1
```

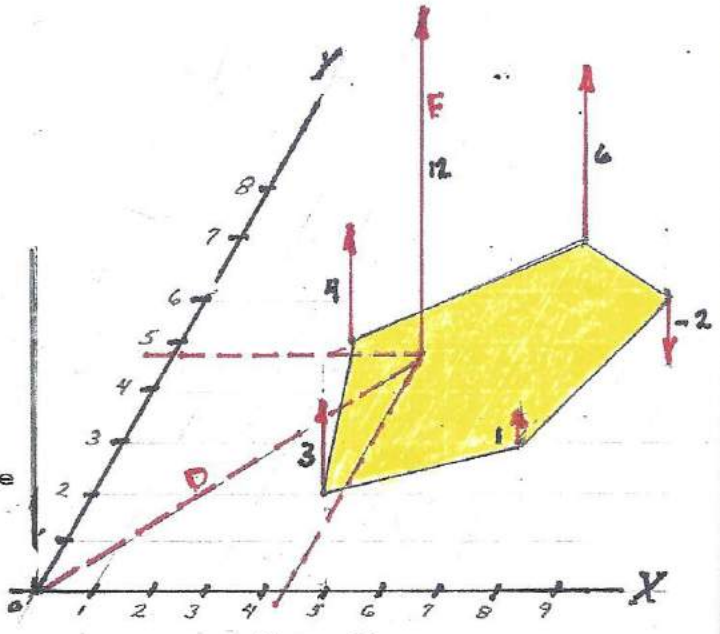


Figure 14

13 Polygon Space Walk

In the preceding example, the polygon was defined by specifying its vertices.

Alternatively, one could have started with an arbitrary point, sp, such as sp=. 4j2 and apply the monad, RS=. +/\, to a series of points, S, and proceed sequentially from sp to the next point in S, with the objective of returning to the starting point. This is equivalent to a "space walk", SW=. RS sp,S which is a complex vector in which its items are the vertices of the polygon and the monad, EC=. {.-{:, applied to SW returns any error of closure.

Two examples are illustrated: Case 1 produces the polygon of this example. Case 2 injects an error in S to create the error which is detected by verb EC=. {.-{:

Case 1

```
   S=. c=. 3j1 1j3 _2j1 _3j_2 1j_3

   Show  ". 'SW=. RS sp,S'
```

4j2	7j3	8j6	6j7	3j5	4j2

```
   EC SW
0
```

Case 2

```
   S=. co=. 3j1 1j3 _2j1 _5j_3 1j_3
   Show  ". 'SW=. RS sp,S'
```

4j2	7j3	8j6	6j7	1j4	2j1

```
   EC SW
2j1
```

The verb RS (Running Sum) has applicability in surveying to detect and calculate the error of closure of polygons. If the points are real numbers, (imaginary part of SW=0), it applies to maintaining a balanced checkbook wherein positive numbers are associated with deposits, and negative numbers associated with withdrawals.

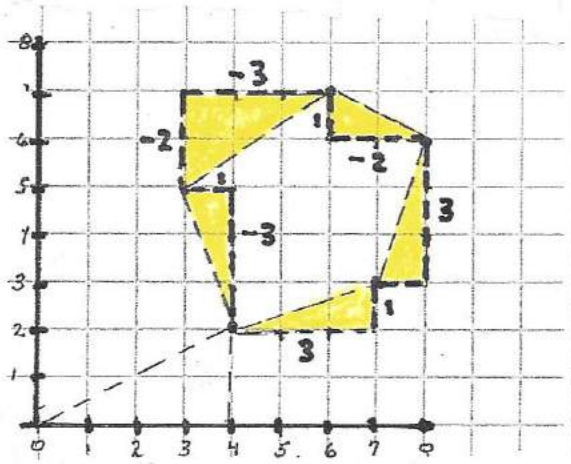


Figure 15

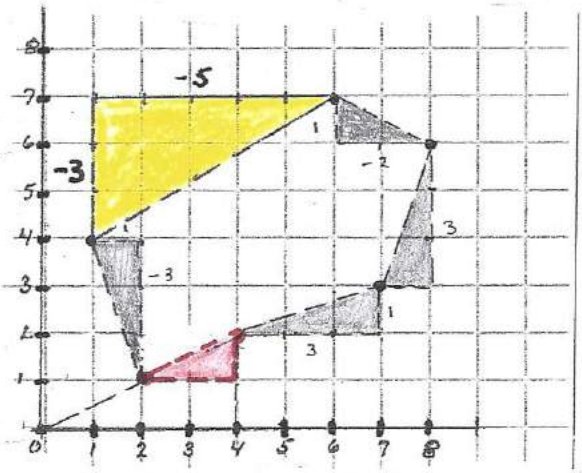


Figure 16

Appendix

Comment

To conserve space in this document, the print precision for all calculations was changed from the default value of 6 to 4 using monad pps=. 9 !: 11.

User-defined verbs not defined in body of text

```
Ea=. &>
Ip=. */ . *
Det=. -/ . *
```

```
ang=. {:@*.
r2d=. %&1p1@*&1806
deg=. r2d@ang
```

```
Show=. <&>
```

Reference

Although there is a considerable amount of information on the technical achievements of Georg Alexander Pick, his life, and the stress he endured, has become increasingly appreciated.

He was born in Vienna on October 10,1859 and was educated at home by his father until he was eleven. At age sixteen he entered the University of Vienna and received his doctorate at age twenty-one. He then became an assistant to Ernest Mach, who was twenty-one years his senior, at the Charles-Ferdinand University of Prague. Six years later, he took a year's leave to study under Felix Klein at the University of Leipzig, returning to Prague where he remained until his retirement. He was forty when he published what is probably his most universally known paper on recticular geometry, or what is now more generally referred to as "lattice geometry". In it, he derived the relationship between lattice lines and points to the area, A, of a lattice polygon and showed it to be A = x + (y/2) -1. The lattice points, x and y were defined previously and this relationship is expressed in J as the dyad "Pick"

In 1905, Einstein, who was 26 and 20 years Pick's junior, and working in a patent office in Zurich, Switzerland, published his Special Theory of Relativity. Pick, became aware of it and, in 1911, as dean of the philosophy faculty at what by that time had become known as the German University of Prague, was the driving force acquiring Einstein as chair of mathematics and physics. Once together, they became close friends and shared a mutual interest in music and science until Einstein's departure to Zurich in July of 1913. During these years, Pick, upon learning of Einstein's struggle to formulate his ideas of general relativity suggested that the newly emerging notions of tensor calculus by Ricca and Levi-Civita might be useful, as indeed it was!! In 1915, Einstein published his General Theory of Relativity.

In 1927 Pick retired as professor emeritus and returned to Vienna, and 1933 Einstein immigrated to the United States. In March 1938, Hitler invaded Vienna, and in March 1939, Hitler invaded Prague. On July 13, 1942, Pick, who was one month short of 83, was sent to the Thereisenstadt Concentration Camp where he died two weeks later.

Shown below is an example of a polygon and the use of Pick's area formula taken from "Pick's Formula: A Retrospective", by Duane DeTemple, Mathematics Notes from Washington State University, Vol.32 Nos. 3 and 4.,November 1989.

The polygon has x= 3 lattice points in its interior and y=9 lattice points on its boundary.

```
   'x y'=. 3 9
```

Pick proved that:

```
   ]Area=. x+(y%2)-1
6.5
```

Alternatively, this relationship[can be expressed as the dyadic hook named in his honor:

```
   PICK=. + [:<:-:
   ]Area=. x PICK y
6.5
```

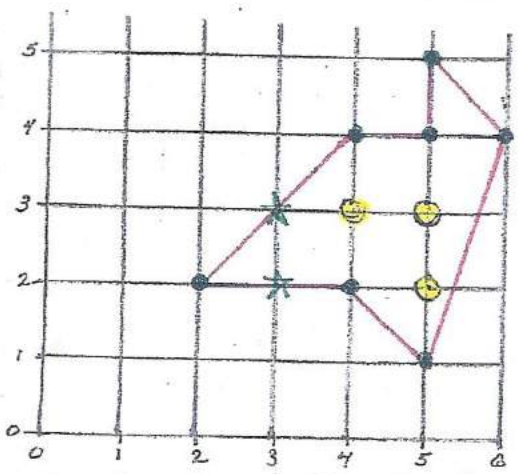


Figure 17