Language Learning Diary - Part Two

Linas Vepstas

2021-present

Abstract

The language-learning effort involves research and software development to implement the ideas concerning unsupervised learning of grammar, syntax and semantics from corpora. This document contains supplementary notes and a loosely-organized semi-chronological diary of results. The notes here might not always makes sense; they are a short-hand for my own benefit, rather than aimed at you, dear reader!

Introduction

Part two of the diary on the language-learning effort starts with the new task of of closed loop learning. The idea of closed loop learning is that the accuracy of the learned grammars can be very closely monitored and measured, thus allowing the learning algorithms to be tuned for speed and measured for accuracy. The closed loop is a basic three-step process:

1. Generating random but controlled grammars
2. Generate a text corpus from these grammars
3. Learn a new grammar from the corpus
4. Compare the controlled-grammar to the learned-grammar
5. Tune algorithms and procedures, and repeat.

The step-by-step instructions can be found in the file "README-Calibration.md".

February 2021

Restart the project, finally making some headway. ToDo items:

- Describe how the "uniform random sampling of sentences" is performed.
- Enable weighted random sampling of sentences.
- Fix multi-sense bug in gen-dict.scm circa line 85

First run. Instructions in "README-Calibration.md". Lots and lots of bugs fixed and lots of pipeline was automated.

March 2021

First real experiment, in expt-7/expt-8/expt-9. The dict in expt-7 fails to generate the correct corpus, because the generator does not expand synonyms (this is a combinatorial explosion and it just doesn't do that.) Expanded by hand in expt-8. Same data, new scripts and config in expt-9.

Issues:

- After MST parsing, the grammar is correct, in that (I think) it will produce exactly the same corpus. However, the rules are different (and more verbose). TODO: check that the same corpus is actually produced. How? Answer: generate the corpus, and compare...
- After MST (MPG) parsing, the verbs link to obj-determiner instead of object. Why? Was there a cutoff that was missed? The corpus is just .. tiny.
- After (gram-classify-greedy-discrim 0.5 4) the right clusters are produced, but the connectors are not clustered; they need to be. TBD.
- Result: the correct corpus is produced (via manual checking), however, extra sentences are produced, which are missing the verb. This is because the wall links to the two determiners, and the wall can be skipped. However, the sentence with the verb has a higher MI. (11.51 instead of 8.92. See below.) This is due to a bug. See below.

Here's the result:

```
linkparser> the squirrel a dog
Found 1 linkage (1 had no P.P. violations)
        Unique linkage, cost vector = (UNUSED=0 DIS=-8.92 LEN=2)

     +-----------TB-----------+
     +---TB---+---TE---+      +-TE-+
     |        |        |      |    |
LEFT-WALL.2 the.1 squirrel.3 a.1 dog.3

linkparser> the squirrel saw a dog
Found 1 linkage (1 had no P.P. violations)
        Unique linkage, cost vector = (UNUSED=0 DIS=-11.51 LEN=4)

     +--------------TB--------------+
     |                 +-----TC-----+
     +---TB---+---TE---+---TF--+-TD-+-TE-+
     |        |        |       |    |    |
LEFT-WALL.2 the.1 squirrel.3 saw.4 a.1 dog.3
```

Hypothesis: The MI of wall-verb is lower than the MI of wall-determiner. Thus, the planar parser always picks the wall-determiner. Lets find out.

Heh. There is no MI wall-verb! Ouch. This was due to bad sampling; fixed github.com/opencog/opencog commit 895226228 Mar 16 2021. Sheesh.

Expt-10

expt-10, this is fixed. The parse trees are now very rich. Generated sentences:

- length 4 - none
- length 5 - the expected ones.
- 6 - LEFT-WALL LEFT-WALL plus valid sentence
- 7 - none
- 8 - LEFT-WALL the mouse saw the dog chased a bird
- 9 - double left wall
- 10 - the LEFT-WALL a cat saw the dog chased a squirrel – and also a triple-left-wall.

WTF. what's with the crazy multi-left-wall!? Heh. Here we go:

```
     +---------------------TI-------------------+
     +------------------TB-----------------+    |
     |           +------TI-----+           |    |
     |           +---TB--+--TC-+--TO--+-TE-+-TC-+
     |           |       |     |      |    |    |
LEFT-WALL.2 LEFT-WALL.2 a.1 mouse.5 saw.3 a.1 cat.5
```

Expt-11

So... expt-11 places a period at the end of every sentence. That terminates the infinite-recursive lengths being generated to only finite-length sentences. There is a total of three different parses. All have exactly the same cost. These are as follows:

```
Found 3 linkages (3 had no P.P. violations)
Linkage 1, cost vector = (UNUSED=0 DIS=-15.67 LEN=10)

     +-------------------TH-------------------+
     |              +------------TJ-----------+
     +------TG------+              +----TE----+
     +---TF---+--TD-+---TI--+--TC--+--TD-+-TJ-+
     |        |     |       |      |     |    |
LEFT-WALL.2 the.1 dog.3 chased.4 the.1 cat.3 ..5
```

```
      Linkage 2, cost vector = (UNUSED=0 DIS=-15.67 LEN=11)
    +-------------------TH-------------------+
    |               +------------TJ-----------+
    +------TG------+------TB------+----TE----+
    +---TF---+--TD-+---TI--+--TC--+--TD-+-TJ-+
    |        |     |       |      |     |    |
LEFT-WALL.2 the.1 dog.3 chased.4 the.1 cat.3 ..5

         Linkage 3, cost vector = (UNUSED=0 DIS=-15.67 LEN=12)

    +-------------------TH-------------------+
    |               +------------TJ-----------+
    |               |       +--------TK-------+
    +------TG------+       |      +----TE----+
    +---TF---+--TD-+---TI--+--TC--+--TD-+-TJ-+
    |        |     |       |      |     |    |
LEFT-WALL.2 the.1 dog.3 chased.4 the.1 cat.3 ..5
```

Notable in the above:

- Only one parse, the third one, links the main verb to a wall. And then its the right wall, not the left wall.
- The determiners seem to be over-linked, and judged to play a too-important role.
- The output grammar is much more highly detailed and constrained than the intended grammar.

Questions:

- What happens if there are a lot more verbs? Would this make the determiners less important, more important, or have no effect? (My guess is “no effect”)
- To downgrade the importance of determiners would seem to require having sentences without them in it.

TBD:

- Waiting on completion of link-generator so that multiple-sense corpora can be generated. (enabled in lg pull req #1175) Or something like that ... what is the right strategy here? Need to rethink to avoid combinatorial explosion, while also verifying category contents.
- Fix bug to allow multiple-sense word definitions in multiple dict locations.
- Dict generation should auto-handle placing a period at the end of the sentence.

Expt-12

Start work on a single-sense random dictionary. Hit assorted issues with the scripts.

Results:

- non-classified dict has 134391 disjuncts, 11 uni-classes including left-wall. These are raw disjuncts.
- classified dict (i.e. that on which grammatical classification has been performed) has 11286 disjuncts

Time to generate 50 sentences, and all possible sentences, in seconds.

length	time for 50	time for all	num sents
3	3	3	108
4	3	3	779
5	4	4	7107
6	5	8	67935
7	13	31	673812
8	43	370	6855920
9	140		
10	638		
11	963		
12	2180		
13	3364		
14	5850		

Data is semi-meaningless, scripts were broken, data processing would start before data was fully loaded. Try again. Upon restart, the number of sentences generated is order of magnitude lower. Presumably due to corrected clustering; above clustered incomplete lists of disjuncts and thus over-generalized.

Expt-13, expt-14, expt-15

Try again with the same initial corpus. Expt-13 overflowed with fake warning message, so I couldn't see the log; thus expt-14 is an exact rerun. "Exact" in the sense of using the same config. However, the random sampling of pairs during pair counting was different.

Then expt-15 uses exactly the same corpus, with a period at the end of sentence placed manually.

Columns:

- length: length of sentence
- time to generate all sentences, in seconds (expt-13)
- num: number of sentences generated (expt-13)

- expt-14: number of sentences generated
- corpus: number of sentences in input corpus. Capped at 25K sentences for the longer sentences. Second number is how many could have been generated.
- expt-15: redo, but with a period at the end of the sentence.

length	time for all	num expt-13	expt-14	corpus	expt-15
3	3	19	21	10	24
4	2	142	163	23	104
5	2	1130	1356	75	485
6	2	9732	12090	254	2294
7	5	86872	111633	892	10845
8	13	794320	1054583	3402	51673
9	143	7393748	10134151	12728	248242
10	2558	69781807	98702133	25000/48364	1198418
11				25000/187541	5807783
12				25000/733525	28246686

Expt-13 and expt-14 differ only in how the pair-counts were collected (they are randomly different samplings of pairs). Both wildly over-sample the corpus.

The expt-13/14 input corpus lacks periods at the ends of sentences. This seems to be the most likely explanation for the over-generation; i.e. last time a period was lacking, the same thing happened. So expt-15 takes exactly the same corpus - identical copy, and adds a period. This does sharply cut down on the number of sentences, especially the long ones, but still over-generates.

General processing stats:

	expt-14	expt-15
time, pair-counting	109 minutes	107 minutes
pair aid	236/293	259/318
pair dimensions	11 x 10	11x10
pair counts	27367456	29988408
pair sparsity	0.0	0.0
pair entropy	5.96=2.90+3.06	6.14=3.08+3.08
pair MI	0.0014	0.012
time, mpg-parsing	12 minutes	19 minutes
mpg aid	246691	201594/298813
mpg dim	11 x 111712	12 x 97203
mpg counts	1067417	1159801
mpg sparsity	3.19	3.49
mpg MM^T support	134663	104049
mpg MM^T count	2341493237	9220550155
mpg MM^T entropy	2.95	0.232
gram dim	3 x 103552	4 x 92299
gram count	880090	1080580
gram sparsity	1.60	2.46
gram entropy	11.58=11.30+1.09-MI	8.83=8.75+1.15-MI
gram MI	0.806	1.08
dict records	102644	67140

Issues:

- There seems to be a dataset issue: both the disjunct pairs and gram pairs have 1/3rd of them without counts on them. .. They are not being saved, after clustering (clustering causes deletion of many disjuncts, and alteration of counts on all disjuncts.) I'm guessing this failure results in bad MI's? Anyway, its a bug is fixed in the new clustering shell scripts.

After above fix, verify export. gram-1 is a re-export of the original expt-13 run, while gram-4 is export of the fixed run. (Both start with the same disjuncts. I think there's nothing stochastic/random during processing, so it should be repeatable...)

	expt-13/gram-1	expt-13/gram-4
gram dim	3 x 103325	4 x 111320
gram count	881340	939886
gram sparsity	1.60	1.92
gram entropy	11.57=11.29+1.09-MI	11.89=11.61+1.17-MI
gram MI	0.806	0.894
dict records	101922	117807

So .. similar but not the same. How about sentence generation? expt-13-gram-1 and expt-13-gram-1a are identical (the 1a version is from a re-export; so we're exporting the same stuff).

The classes in gram-1 are <b f> <e j> and left-wall. The classes in gram-4 are the same plus <i#uni> ... there was no word i in gram-1 !! Wow, that's a big drop.

length	time for all	expt-13-gram-1	expt-13-gram-4
3	3	19	19
4	2	142	163
5	2	1130	1414
6	2	9732	13147
7	5	86872	125527
8	13	794320	1225346
9	143	7393748	12156101
10	2558	69781807	122141737

OK, so the numbers are dramatically larger. Apparently, this is due to the previously dropped word i. Yikes!

Well, the above is massively under-counting – it is only sampling one random word-draw per class. Since multiple words are in each class...

... anyway, this is nuts, because the connectors need to be classified, instead of issuing new connector types. So more work before something meaningful is possible.

expt-13 vs expt-15 precision, recall

So now that we've got things working, lets look at precision and recall. Clearly precision will be terrible, but maybe recall will be excellent? Compare expt-13-gram-4 to expt-15-gram-2 (which rebuilds after fixing the borked save.

Uhh .. No its not working yet, in the sense that the connector classes are being mis-handled. Those need to be grouped correctly before export. More work...

expt-16

Due to absence of left-wall in the above, breaking the dict-compare step, tried again, generating a new dict with walls (after manually adding a wall to the dict of expt-15, and ending punctuation to the dictionary.) Well ... clustering worked quite differently. Here's a summary.

Pair counting seems to be more-or-less the same, slightly higher MI=0.030 which is still minuscule. MPG entropy, counts etc. look similar to the earlier runs.

Gram classification: only 2 words assigned to the same class. Oh, this used the "disinfo" classifier, whereas the earlier runs used the "fuzz" classifier. That could account for everything, I guess. Lets take a look.

	disinfo 3.0 4	discrim 0.5 4	fuzz 0.65 0.3 4
gram dim	11 x 75552	7 x 75667	7 x 75667
gram count	1011444	951215	960646
gram sparsity	3.37	2.77	2.77
gram entropy	11.48=10.01+3.15-MI	10.38=9.94+1.90-MI	10.45=9.91+2.06-MI
gram MI	1.68	1.46	1.52
dict records	80460	77735	77750

Clearly, the resulting clusters are sensitive to the parameters controlling classification. The above parameters seemed reasonable for the large English dataset. They may be unreasonable here!? But this is very unclear.

The MI's are larger, across the board (vs. 0.8 or 0.9 before.) How about sentence generation? We expect disinfo to be more accurate, since it did very little clustering.

length	time disinfo	disinfo	time discrim	discrim
3	61	60	3	18
4	67	523	3	107
5	131	5217	3	752
6	137	51368	3	5371
7	183	488376	5	37923
8	715	4514440	8	268248
9			37	1909447
10			262	13616410

Again, the discrim is under-counting, because of more categorization.

Next step: fix the conjoined clustering, with shapes.

Wow. So MM^T entropy with shapes is 5.03 which is huge compared to the dj-only MM^T so it really is something new and different! With shapes, and with gram-disinfo, there were no merges.

	disjunct disinfo 3.0 4	shape disinfo 3.0 4
MM^T MI		5.03
gram dim	11 x 75552	12 x 75552
gram count	1011444	1015356
gram sparsity	3.37	
gram entropy	11.48=10.01+3.15-MI	11.50=10.02+3.18-MI
gram MI	1.68	1.69
dict records	80460	80807

OK, so looks like shape created no categories at all. So how does that work out for generation?

```
link-generator -l learned -c 123123123 -s 3
```

length	corpus	time for all	expt-16-shape
3	4	38	61
4	21	69	566
5	50	147	5638
6	179	124	55546
7	621	172	531075
8	2246	642	4937036
9	8850		
10	> 25K		

OK, so wildly over-generating sentences, despite effectively no clustering being done. Didn't we do an experiment without clustering?? I can't find it above. Why are we over-generating? How to best explain it? Too small a vocabulary?

- Issue 1: given a fake-lang the generator is failing to generate all possible sentences. Fixed in link-grammar pull req #1175
- Issue 2: there are "accidental" synonyms cause of 1 above: many POS'es are shared in common between many words but are not completely sampled, thus creating "accidental synonyms".

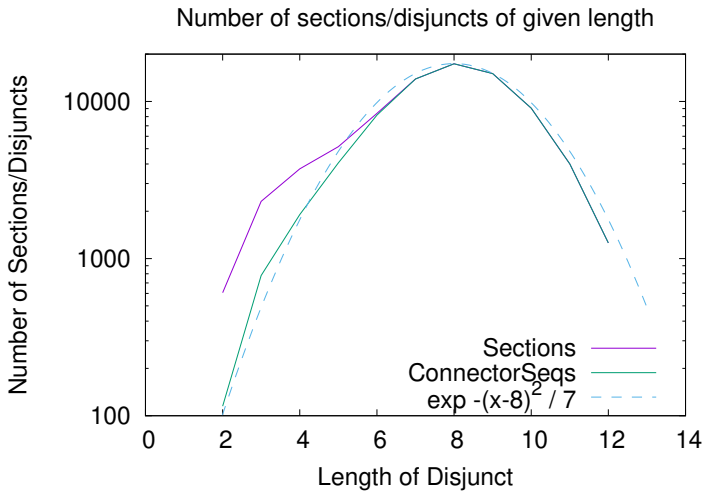
OK, so back to that one, except this time, its in the shape of a bug...

Sections and disjuncts

The Sections that were learned in expt-16 have a surprising number of connectors on them, averaging at 7.7 connectors per section. This seems way too large. What's up with that? Step one: get a more detailed view.

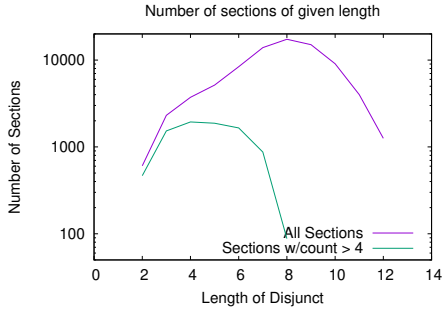
```
# column 1 length
# column 2: number of Section
# column 3: number of ConnectorSeq
#
1  0  0
2  606   115
3  2311  780
4  3723  1904
5  5150  4066
6  8387  8183
7  13907 13896
8  17362 17362
9  15043 15043
10 9061  9061
11 4000  4000
12 1257  1257
13 0
```

So ... long disjuncts appear on one and only one word (as witnessed by identical counts for length 8 and above). Short disjuncts might be shared with multiple words. This implies that words are not mergeable, or rather, the distinct long sections are preventing merger. Here a graph of above.



So its very Gaussian, both peak and tails. All these gaussians here and in earlier results imply that central-limit theorems hold. And all kinds of other classical theorems should hold. Have not been leveraging those theorems, so far. If we did, what would we get? Note it's Gaussian despite the fact that negative connector-seq lengths don't make sense! Right? What would a negative connector-seq length be? A repulsion? an anti-connect statement? "Must never connect"?

What happens if we exclude Sections with a low observation count?



The above shows all sections, and sections that were observed 5 or more times. Apparently, the long, complicated disjuncts are observed very rarely.

How should this be interpreted? Common sense seems to suggest that low-count observations are "noise" and should be cut before any merging is performed. Doing so will certainly increase the similarity of vectors. But are cuts really needed? Perhaps

the similarity measures can already deal with these? If so, then the only reason for cuts would be performance, rather than accuracy.

How do these affect cosine-similarity vs MI-similarity? Clearly, for cosine-similarity, low counts means short vector components, and so these will not contribute much to the dot product. Likewise, the MI-similarity is built on a dot-product, so again, these should not contribute much. Even more-so, since the MI-similarity never takes a square-root of the dot product. It does, however, re-weight basis elements in a fashion that I do not yet have a good intuition for. (Or rather, developed an intuition, and now I've forgotten what it was... Hmmm.)

TODO: The Shapiro–Wilk test can be used to determine how close a distribution is to a Gaussian. The Wikipedia article on it is as clear as mud.

April-May 2021

Before starting expt-17 with fixed sampling, it is time to ponder connector merging. This is turning out to be non-trivial. The general upshot of the below is that the original 2019 concept for Shapes and CrossSections is a good, strong idea. However, assorted non-commutativities arise, and how to resolve them appropriately is not entirely clear. Thus, we embark on a journey of discovery...

But first: connector merging is needed because without it, the clustering fails to adequately reduce the size of dictionaries. The meta question is this: if words A and B are determined to be near-synonyms, and assigned to the same cluster, then what should be done with connectors that have A or B in them? Should all such connectors be automatically replaced by the cluster? The naive answer is, "yes, they should be".

The less-naive response is "but multiple word-senses". That is, A and B are not just words, but are word-vectors, and vector B may be a linear combintation of two different word senses. One of these senses might be synonymous to A, and the other might be completely diffferent. Thus, we want to merge that part of vector B that is (nearly) colinear with A, while leaving behind a different vector B-prime that is associated with some other word-sense for the word B. When encountering B in a connector, is that B in the sense of wordclass-AB, or is it in the sense of B-prime? If the former, then obviously, that connector should be updated to use wordclass-AB; otherwise, it needs to be left alone, thus implicitly defaulting to B-prime.

TBD: Writing the above, it appears that the need for an explicit B-prime has been overlooked. This could cause havoc, in allowing inappropriate linkages! This is important, and needs to be addressed ...

Connector merging

Some of the questions that arise with connector merging, to figure out:

- How would connector merging affect clustering results?
- If connectors are merged, then how should vectors be handled? (Naive vectors no longer work, because the basis is now different.)

- What is the best or most correct merge algorithm?

Lets look at a toy model. Suppose the dict is

```
A: C+ & D+
B: C+ & D+

C: A- & E+
C: B- & F+

D: A- & G+
D: B- & G+
```

From this, conclude that A and B can be merged. However, the connectors on C cannot be merged, the connectors on D can be merged. The final dict is then (with some abuse of notation)

```
<wclass-AB>: C+ & D+
A B: <wclass-AB>

C: A- & E+
C: B- & F+

D: <wclass-AB>- & G+
```

Next suppose that we have

```
S: C+ & D+
T: C+ & D+

C: S- & J+
C: T- & F+

D: S- & H+
D: T- & G-
```

The determination to merge is now trickier. Naively, S can be merged into `<wclass-AB>` but doing so would wreck the connector set on D, as it implies an `(S- & G+)` which was not observed, and thus not mergeable. ... unless we are willing to create such new unobserved cases.

Merging T looks OK.

Perhaps this looks bad because we are not doing shape vectors. The shape vector for the above would be:

```
A: (C+ & D+) or (C:x- & E+) or (D:x- & G+)
B: (C+ & D+) or (C:x- & F+) or (D:x- & G+)
```

```
C: (A- & E+) or ...
C: (B- & F+) or ...

D: (A- & G+) or (A:C+ & x+)
D: (B- & G+) or (B:C+ & x+)
```

So the decision to merge A and B is less clear-cut, when using shapes (they are not perfect synonyms). The shape variant of the S thing is

```
S: (C+ & D+) or (C:x- & J+) or (D:x- & H+)
T: (C+ & D+) or (C:x- & F+) or (D:x- & G+)

D: (S- & H+) or (S:C+ & x+)
D: (T- & H+) or (T:C+ & x+)
```

So using shapes weakens the decision to merge S into AB. The decision to merge T remains strong. So it seems that shapes do offer a stronger foundation on which to make merge decisions. They examine similarity (almost-synonymy) out to a greater distance from the germ. It seems like they also allow things to continue to be treated as vectors, instead of muddling the concept of vectors. After the AB-merge[1] the result would be:

```
<wclass-AB>: (C+ & D+) or (C:x- & E+) or (C:x- & F+) or (D:x- & G+)
A B: <wclass-AB>

C: A- & E+
C: B- & F+

D: (<wclass-AB>- & G+) or (<wclass-AB>:C+ & x+)
```

Now, given this merged-AB thing, when happens when we look at merging S? Well, S is meh, T looks better. The vector for T is comparable to the vector for <wclass-AB> so vector similarity works for that merge decision.

Conclude: the original plan from a few years ago works and holds water. Use shape vectors for merge decisions. Once this is done, connectors can be swept up.

Non-commutativity

The above description pulls a sleight-of-hand, which presumes an algorithm that is able to crawl across individual disjuncts, compare them, and update words with merged word-classes. Such an algorithm can be written (and has been written/prototyped). It leads to some confusion, because the shapes/cross-sections are no longer consistent. Lets call the above the "connector sweep algorithm", or "sweep" for short.

[1] When using the "union-merge" strategy, as described in in `src/gram-projective.scm`. In practice, the merge style used is typically `merge-project`, which would accept only a fraction of `(C:x- & E+)` and `(C:x- & F+)` into the final vector.

Starting with the above example:

```
<wclass-AB>: (C+ & D+) or (C:x- & E+) or (C:x- & F+) or (D:x- & G+)
```

the sections can be reconstructed from the cross-sections. The reconstructed sections are

```
C: <wclass-AB>- & E+
C: <wclass-AB>- & F+

D: <wclass-AB>- & G+
```

Comparing, the reconstructed section on D is the same as what the sweep algo produced, but the sections on C are *not* what the sweep merge offers. That is, the sweep is not commutative with the creation of shapes. This is a problem for maintaining the consistency between sections and cross-sections as clusters grow. The lack of consistency will cause merge judgements to diverge...

Thus, we have at least two algorithms:

- Sweep-merge, as described above, where connectors are replaced by merged-connectors if and only if the the rest of the connector set is identical. This merge algorithm is naively described, since it does not explain what to do if there are multiple connectors in a conector set that might be merged. It's also naive in that it does not explain how counts (frequencies) are to be handled.
- Reshape-merge, which performs the basic projective merge on the germ-vectors, and then reconstructs Sections from CrossSections, thus restoring consistency between sections and cross-sections. It violates the intuitive correctness of the sweep-merge, but only perhaps because the sweep-merge, as naively described above, assumed the "union-merge" strategy of transfering observation counts for vectors taht are not perfectly colinear. The projective-merge count transfers recognize the non-colinearity, and obtain cluster centroids through weighting formulas.

The above ruminations suggest that reshape-merge enjoys an advantage over sweep-merge, as it keeps the section/cross-section duality consistent.

Anyway, the original 2019 plan for using shapes seems to have been a good plan.

Non-Commutivity, Again

The non-communtivity can be hightened with a slightly richer example. Consider

```
A: (P+ & Q+) or (R+ & S+) or (K- & B+)
B: (P+ & Q+) or (R+ & S+)
C: B- & T+
```

The dictionary entry of C is present to remind us of the fact that, if B+ appears as a connector, then B- must also appear as a connector, somewhere.

Based on the first two sections, a decision might be made to merge, with the count on `(K- & B+)` being small enough that it does not disrupt the merge decision. Expanding this into it's CrossSections, the full vectors are:

```
A: (P+ & Q+) or (R+ & S+) or (K- & B+)
B: (P+ & Q+) or (R+ & S+) or (K- & A:x+) or (C:x- & T+)
```

The merge result is then

```
<wclass-AB>: (P+ & Q+) or (R+ & S+) or (K- & B+)
<wclass-AB>: (K- & A:x+) or (C:x- & T+)
```

The cross-section leads to a reconstruction (reshape) of

```
A: K- & <wclass-AB>+
C: <wclass-AB>- & T+
```

How is this to be interpreted? Let's explore some "common sense" reasoning.

Case A: The count on `(K+ & B+)` is so small that it is considered to be noise, and is completely dropped before merging even starts. In this case, A and B are exactly colinear (are exact synonyms). The (naive) merge of A and B is completly unproblematic, except that it leaves C without the ability to connect to anything. This can be handled in one of two ways. One way is to notice that, by detailed balance, the counts on this particular C section must also be tiny, and so this section can be dropped from the dictionary.

Another way to avoid this dangling-connector problem is to presume that the dictionary also includes

```
D: L- & B+
```

which would provide a place for that bar B to connect. but if this were the case, then we did the cross-sections on B wrong. Fixing these would have given

```
<wclass-AB>: (P+ & Q+) or (R+ & S+) or (K- & B+)
<wclass-AB>: (K- & A:x+) or (C:x- & T+) or (L- & D:x+)
```

which then reshapes to

```
A: K- & <wclass-AB>+
C: <wclass-AB>- & T+
D: L- & <wclass-AB>+
```

This is now fully linkable, and there are no dangling pure-B connectors.

In conclusion, this seems self-consistent eitehr way: either we can drop the `A:(K+ & B+)` section entirely, and, by detailed balance, we can drop `D:(L- & B+)` also; or we can keep both, and doing it correctly leaves nothing dangling.

Note that we have to be careful with tracking in the merge algo: when reshaping to get the `C:<wclass-AB>- & T+` section, we have to be careful to notice that the `C:B- & T+` section was a donor, and so it should be removed (its counts driven to zero). Otherwise, C would end with both these sections on it, and it would be a bit wonky.

Case B: The count on `(K+ & B+)` is small but not ignorable. It is small enough to not block the merge decision. There are two issues to resolve. The first is easy: the `C:B- & T+` section should be recognized as a donor to `C:<wclass-AB>- & T+`, and removed (its counts driven to zero). This is easy enough to determine at the time of the merge.

The more difficult issue is what to do about the `<wclass-AB>:(K- & B+)` section, which appears to have a dangling B connector, and the reshape of `A:(K- & <wclass-AB>+)`, which seems to be a double-count. The first leaves a dangling A, the second leaves a dangling B. It seems fairly clear that these should be harmonized, merged together, to give `<wclass-AB>:(K- & <wclass-AB>+)`. It seems that this can be reasonably infered and performed at the time of creation, since the donors are readily identified.

Case C: The count on `(K+ & B+)` is large, large enough to split. That is, A should be understood to be the direct sum of two distinct word-senses, with one word-sense being `<wclass-AB>` and the other being `<A-prime>:(K+ & B+)`. So, if were were able to be absolutely sure that A-prime was really a distinct word-sense, then we should transfer none of the counts from the originating section `A:(K+ & B+)` to the `<wclass-AB>` section.

So, starting with the vectors

```
A: (P+ & Q+) or (R+ & S+) or (K- & B+)[n]
B: (P+ & Q+) or (R+ & S+) or (K- & A:x+)[n] or (C:x- & T+)
```

where square-bracket-n is the count on that section, we create a merge result of the form

```
<wclass-AB>: (P+ & Q+) or (R+ & S+)
<wclass-AB>: (C:x- & T+)
<A-prime>:  (K- & B+)[n]
B: (K- & <A-prime>:x+)[n]
```

The cross-sections leads to a reconstruction (reshape) that appears to be self-consistent, so I don't see any problems here.

If we were to split [n] into some fractional parts, then this would reduce to a combination of case B and the current case, so that should also work.

Connector counts

(TBD, this section needs to be harmonized with the new text above... the below was written before the above was rewritten...) Lets go through above exercise, this time with counts. Suppose the dict, with observation counts in square brackets, is

```
A: C+ & D+ [na]
B: C+ & D+ [nb]

C: A- & E+ [nca]
C: B- & F+ [ncb]

D: A- & G+ [nda]
D: B- & G+ [ndb]
```

The shapes, with counts, are

```
A: (C+ & D+)[na] or (C:x- & E+)[nca] or (D:x- & G+)[nda]
B: (C+ & D+)[nb] or (C:x- & F+)[ncb] or (D:x- & G+)[ndb]

C: (A- & E+)[nca] or ...
C: (B- & F+)[ncb] or ...

D: (A- & G+)[nda] or (A:C+ & x+)[na]
D: (B- & G+)[ndb] or (B:C+ & x+)[nb]
```

Merging A and B, with 100% of count transfer, gives

```
A B: (C+ & D+)[na+nb] or (C:x- & E+)[nca] or (C:x- & F+)[ncb] or (D:x- & G+)[nda+ndb]

C: (A- & E+)[nca] or ...
C: (B- & F+)[ncb] or ...

D: (A- & G+)[nda] or (A:C+ & x+)[na]
D: (B- & G+)[ndb] or (B:C+ & x+)[nb]
```

Looking at the connectors on D, we see that they are mergable, and that the counts are consistent. So, based on this toy model, we can either try to merge connectors directly, or we can, at a later date, merge connectors by reconstructing them from merged shapes. Doing it either way should give the same counts: the operations are commutative.

This is not entirely obvious. It seems to work for the toy example. It seems like the toy example could be converted into a full proof. Yet ... are we missing something? Best bet is to write the code both ways, and very numericall that the operations are commutative.

Fractional counts

Lets try again, this time with fractional counts. Suppose that instead of merging 100% of B into A, we merge only a fraction $0 \leq y \leq 1$ of the count. This gives

```
A B: (C+ & D+)[na+y*nb] or (C:x- & E+)[nca] or (C:x- & F+)[y*ncb] or (D:x- & G+)[nda+y*ndb]
B: C+ & D+ [(1-y)nb] or (C:x- & F+)[(1-y)ncb] or (D:x- & G+)[(1-y)ndb]

C: (A- & E+)[nca] or ...
C: (B- & F+)[ncb] or ...

D: (A- & G+)[nda] or (A:C+ & x+)[na]
D: (B- & G+)[ndb] or (B:C+ & x+)[nb]
```

Then, apparently, the counts will be consistent if and only if the same fraction is used when merging connectors.

Connector merging, with counts

To get all of the above correct, there is a series of unit tests. They work well, but one of the more complex ones has become painfully difficult to understand and debug. It is reviewed here. But first, a change of notation to make it more compact:

- The entry `A:(B- & C+)` will be written as `(A, BC)`. Here, the perenthesis denote a pair (a matrix entry). The letter sequence is just the connector sequence with the directional indicators ignored.
- Entries with word-classes, such as `<wclass-AB>:(K- & B+)` will be written as `({AB},KB)`. The word-class is denoted with set-notation curly braces. Similarly, `A:(K- & <wclass-AB>+)` will be written as `(A,K{AB})`.
- Cross-sections, which were written above as `D:(A:C+ & x+)` will be written as `[D, <A, Cv>]`. The angle brackets denote the shape, and the lower-case `v` denotes the location of the variable in the connector sequence. The square brackets just serve to remind that a cross-section is being discussed.
- A property called "detailed balance" is introduced. This is the idea that corresponding sections and cross-sections should have exactly the same observation count on them. Thus for example, given a section `(A, BC)` which was observed N times, one expects that the two cross-sections derived from it, namely `[B, <A, vC>]` and `[C, <A, Bv>]` are also both observed N times each. Prior to any merging, detailed balance holds "automatically" or tautologically, as a trite statement about how counting is done. The goal is that connector merging should preserve detailed balance as a property. It is assumed to be a desirable property, and is enforced in the code and unit tests.
- Counts will be denoted with a lower-case n written in front of the pair. Thus, `n(A, BC)` would be the number of times that `(A, BC)` was observed.

First merge

The troublseome test is '`connector-merge-tricon.scm`'. The relevant portion is as follows. The dictionary is assumed to contain many entries; the troubleseome subset is this:

```
(j, abe)
(f, abe)
```

A decision is made to merge the vectors for `e` and `j`, based on other dictionary entries not shown here. The "projective merge" strategy is used, so that a fraction $0 \leq p \leq 1$ of the count is merged whenever one of the two vectors is missing an entry at a given basis element. In this case, the merge, denoted with an arrow, is

```
none + (j, abe) -> p * ({ej}, abe) + (1-p) * (j, abe)
```

where '`none`' denotes that there is no section `(e, abe)` and so the projective merge was used. That is, the count on `(j, abe)` is reduced to $(1-p)$ of it's earlier value, and the remaining p is transfered over to `({ej}, abe)`. That is, the total counts are preserved. That is,

$$n'(\texttt{\{ej\}, abe}) = pn(\texttt{j, abe})$$

where n' denotes the count after the merge, and the unprimed n is the count before the merge.

From the connector merging discussion above, we conclude that `({ej}, abe)` should be rewritten to `({ej}, ab{ej})`. The count should be as above, that is:

$$n'(\texttt{\{ej\}, ab\{ej\}}) = pn(\texttt{j, abe})$$

The vector on `e` includes the cross-sections

```
[e, <j,abv>]
[e, <f,abv>]
```

These merge in a similar fashion:

```
[e, <j,abv>] + none -> p * [{ej}, <j,abv>] + (1-p) * [e, <j,abv>]
[e, <f,abv>] + none -> p * [{ej}, <f,abv>] + (1-p) * [e, <f,abv>]
```

From detailed balance, we deduce the two new sections

```
(j, ab{ej})
(f, ab{ej})
```

The counts on these are

$$n'(\texttt{j, ab\{ej\}}) = pn(\texttt{j, abe})$$

and

$$n'(\texttt{f, ab\{ej\}}) = pn(\texttt{f, abe})$$

From the connector merging discussion above, we conclude that `(j, ab{ej})` should be rewritten to `({ej}, ab{ej})`. The first identity arrives at the same count as before, so this rewrite appears to be self-consistent. Everything works out.

Second merge

The first merge is more-or-less straightforward. The trouble comes with the second merge. Here, it is decided that the vector for `f` should be merged into `{ej}`. The preservation of detailed balance creates subtleties and ambiguities.

The final count on `({ejf}, ab{ejf})` gets contributions from three sources:

- The starting count on `({ej}, ab{ej})`, which is

  $$n'(\texttt{\{ej\}, ab\{ej\}}) = pn(\texttt{j, abe})$$

  as given above.

- A contribution from `(f, ab{ej})`, which was created in the first merge, via detailed balance from the earlier cross-section `[{ej}, <f, abv>]`. This is merged in it's entirety into the the existing `({ej}, ab{ej})`. The projection merge is

  ```
  ({ej}, ab{ej}) + (f, ab{ej}) -> ({ejf}, ab{ej})
  ```

  This merge absorbs the entire count on `(f, ab{ej})` because `({ej}, ab{ej})` already exists. The contribution is thus $n'(\texttt{f, ab\{ej\}}) = pn(\texttt{f, abe})$. Rewriting then promotes `({ejf}, ab{ej})` to `({ejf}, ab{ejf})`.

- A contribution from `(f, abe)` via the projection merge

  ```
  none + (f, abe) -> q * ({ejf}, abe) + (1-q) * (f, abe)
  ```

  and then the subsequent rewrite of `({ejf}, abe) -> ({ejf}, ab{ejf})`. This contribution is $qn'(\texttt{f, abe}) = q(1-p)n(\texttt{f, abe})$.

The total of these three contributions is then

$$\begin{aligned} n''(\texttt{\{ejf\},ab\{ejf\}}) =& n'(\texttt{\{ej\}, ab\{ej\}}) + n'(\texttt{f, ab\{ej\}}) + qn'(\texttt{f, abe}) \\ =& pn(\texttt{j, abe}) + pn(\texttt{f, abe}) + q(1-p)n(\texttt{f, abe}) \end{aligned}$$

Note that this result is history-dependent: merging j into e first, then f gives a different result than merging f into e, then j (and presumably different than the third possibility, of merging f and j first, and only then adding e).

An open question is whether there is a way of performing the merges that are history-independent, and what would that mean.

Again, additional details are in the test file '`connector-merge-tricon.scm`'.

Connector Merging, Conclusion

After much work: there are ten unit tests, all passing, with the final fix in commit 5e1d7dfb94867f22642d7cdf0621a833bb96092e of 24 May 2021 which fixes a problem not found in the unit tests; it requires a real-world test-case. Need to run `(check-balance LLOBJ)` to evoke it.

expt-19 (May 2021)

Moving on... expt-19 reuses the same corpus as expt-16, and, in order to be comparable to earlier results, reuses the `pair-counts` and the `mpg-parse` disjuncts from expt-16.

Using `(gram-classify-greedy-disinfo psa 3.0 4)`, there wern't any merges that got done. That's because similarity never got above 3.0. The closest that things got was these pairs:

pair	MI-distance
!-i	2.3578
a-i	1.2513
f-c	1.1714
h-c	1.0837
f-h	1.0757
b-g	1.0559
j-b	0.5846
j-g	0.5265
d-f	0.5170
e-b	0.5066
d-c	0.5026
e-j	0.4612
e-g	0.4503
d-h	0.4446
e-d	0.0049

Which is ... pretty distant. The distance !-i is alarming. A core problem is that this is a very mixed grammar: lots of ambiguity, lots of word senses, no particular clean factorization. Just looking at it, its quite cloudy as to the actual structure. In particular, although each word belongs to only one word-class, the word-classes have lots of mixed, shared POS entries, and each POS is a seemingly random (duhh) unstructured mess. For example:

- pos-e has single, divalent, trivalent disjuncts on it. Except for a few words, most of English is not like that.
- pos-c has one disjunct. It's identical to pos-d, which has two disjuncts.
- pos-e has a huge number of disjuncts on it, as do pos-i and pos-j. This seems to allow very grammatically complex sentences to be generated, which "of course" are going to be very hard to decode.

Overall, the grammar appears to be over-complex, and very unlike a natural language grammar. it seems unlikely, just from eyeballing it, that the grammar could be untangled without a very large, exhaustive examination of the corpus. This is a bad experimental base.

Issues:

- Is there any sense in which the word "mixing" is appropriate, in it's technical sense (from ergodic theory?) Can we define mixing from the point of veiw of disjunct ambiguity? Of the indiscernibility of grammars given a corpus?
- Is the automatic grammar generation controlling sufficiently for word-senses? Yes, there's a tunable paramter for that, but some disjuncts accidentally appear in multiple POS, thus making those POS at least partly synonymous.
- Is the current automatic grammar generation API appropriate? It was based on an intuitive sense of factorization, but the randomness seems to eassily generate ambiguous grammars.
- How does one measure the complexity of a grammar?
- Is there some easy way of writing down its factorizability?
- Is there a way of proving that two grammars are equivalent? If two different grammars generate the exact same corpus, then is there some algorithm that can transmute one grammar into the other? How is this found/discovered?
- How can one characterize human natural language grammars? That is, if an artificial grammar is generated, how can we know if it is similar to a natural human language? I don't think the rainbow of human natural languages lines up well (or at all) with the axes of tunable paramters in the grammar generator.

Conclude: the expt-19 grammar is over-complicated, ambiguous, mixed, ugly. We need to restart with a simple grammar.

Also conclude: I do not understand how the ambiguity of grammars works. I do not really understand how factorization works. The artificial grammar generator "works" but I don't understand what it is generating. I don't know how close it is to typical human grammars. There's a bit of a "back to the drawing board" moment here.

expt-20 (May 2021)

Start again, this time with a simple, relatively unambiguous, relatively unmixed grammar. Perhaps even with a tiny artificial subset of English!? Just to make eyeballing easier?