P A T T E R N I Z A T I O N
– tutorial (in progress) –

Alexander Pfaff
nonintersective at gmail.com

August, 2024

Abstract

In the following, we will introduce and illustrate the main functionalities and methods of the `Patternization` tool. A *pattern* as understood within this framework is the linear construal of a syntactic constituent, i.e. as a sequence of category labels. In the current version, "syntactic constituent" $\leftrightarrow$ noun phrase (= NP); "category label" is to be understood relative to the given annotation system. For instance, the following example

(0)	these	two	black	horses	behind the tree	that are grazing
	determiner	numeral	adjective	noun	prepositional phrase	relative clause

could be rendered as a tuple
⇒ (Det, Num, Adj, N, PP, RC), or
⇒ (Det, Num, Adj, N, P, Art, N, Comp, Aux, V) . . . .
depending on whether the underlying system incorporates syntactic categories, or is purely POS-based (POS = “part-of-speech”).

The current version draws on a subset of the material in the NPEGL noun phrase database (https://spraakbanken.gu.se/en/resources/npegl) and the annotation system used there. It comprises data for Old Icelandic, Old English and Old Saxon.

For more (background) information on the DB material, texts, annotation system, and a full overview of the category labels used, please consult Pfaff and Bouma (2024), available here:
– https://zenodo.org/records/10641183/files/436-Bech-Pfaff-2024-1.pdf?download=1;
see also Pfaff (2019). For a more thorough account of the framework and syntactic motivation, see Pfaff (2024), available here:
– https://zenodo.org/records/10641185/files/436-Bech-Pfaff-2024-2.pdf?download=1

NB: This is a prefinal version; some methods have not been (fully) implemented, and the documentation is not complete yet. For feedback, questions, bugs etc. please contact the author.

Patternization emerged in the context of the NPEGL project (NRK grant no.261847; https://www.hf.uio.no/ilos/english/research/projects/noun-phrases-in-early-germanic/).

1 The `Patternize` / database class

1.1 Set-up and Initialization

Download the following files into the same directory:

- `Patternization_main.py`
- `oice_db.json`
- `oeng_db.json`
- `osax_db.json`
- `empty_db.json`

Open & run `Patternization_main.py` –
alternatively: `import Patternization_main` –
in the IDE of your choice.

Create a `Patternize` object with the language of interest being the argument of the constructor (default setting = Old Icelandic):

- `>>> ice = Patternize("oice")` – for Old Icelandic
  NB: this is equivalent to `ice = Patternize()`
- `>>> eng = Patternize("oeng")` – for Old English
- `>>> sax = Patternize("osax")` – for Old Saxon

```
>>> ice
<class 'Patternize'>

>>> print(ice)
[Patternize::Database; Language:  Old Icelandic; Size:  9072]
```

1.2 Basic Features

A `Patternize` object is an enhanced database object (henceforth: *db*), and, in a manner of speaking, a dynamic (→ mutable) datastructure, some properties of which may change or be updated. The following are some core properties of a db:

(1) a. `>>> ice.size` (number of NPs in the db)
b. `>>> ice.ID` (language of the db)
c. `>>> ice.maxPatternSize` (size of the longest pattern)
d. `>>> ice.tokenize` (number of categorized items in the db; = sum of individual NP lengths)

Some properties are initialized with some empty value:

(2) a. `>>> ice.update` (initialized as `()`)
b. `>>> ice.rnd_update` (initialized as `0`)

The property `ice.update` indicates which modifications/filterings have taken place (see below), while `ice.rnd_update` shows how many randomized dbs have been generated (see below).

A `Patternize` object has three attributes that are themselves `Patternize` objects with all the same functionalities; those are:

(3) a. `>>> ice.rnd_database`
b. `>>> ice.prenominalDomain`
c. `>>> ice.postnominalDomain`

⇒ `<class 'Patternize'>`

(4) a. `>>> print(ice.rnd_database)`
b. `>>> print(ice.prenominalDomain)`
c. `>>> print(ice.postnominalDomain)`

⇒ `[Patternize::Database; Language:  Old Empty-ese; Size:  1]`

As the output of the `toString` method shows, however, they are not at the outset initialized; they are empty dbs as indicated by the dummy language “Old Empty-ese”, and must be initialized appropriately (see below).

1.3 Modification / Filtering

As already mentioned, a `Patternize` object is a dynamic db and may be considered a (recursive) "working database". It is possible to modify the db in various ways where a modification can also be thought of as a primitive query:

(5) `>>> ice.filter(cat, present=True)`

The `.filter` method takes two arguments:

(i.) a string representing a category label (e.g. "Dem", "Md.Aj.Lx", "RC" ... for a full overview of labels, see the appendix in Pfaff and Bouma (2024))

(ii.) a Boolean value, with the default setting `True`.

If `present == True`, every NP in the db containing the respective `cat` will be filtered out (= removed). Conversely, if `present == False`, every NP **not** containing `cat` will be removed. The beauty of the procedure is that the output of the `.filter` method can be viewed as the result of a query, in that the remaining db comprises only material with a given specification, but at the same time, it is still a db with all functionalities. That is the db can be examined and processed as is, or the `.filter` method can be called again and the db further pruned. If the full dataset is needed, simply create a new `Patternize` object.

A special filter method with a predefined setting is the `.ndb` method:

(6) `>>> ice.ndb()`

where "ndb" stands for ***n**ominal **d**ata**b**ase*. The specification is such that only NPs will be retained that

(i.) contain exactly one lexical/common noun ("N.C"), and

(ii.) do not contain a coordination structure ("&").

We will use this setting for the rest of this tutorial. Now compare some of the properties mentioned above before and after the method is called:

(7) a. `>>> ice.size`
before `.ndb`: 9072 after `.ndb`: 7981

b. `>>> ice.maxPatternSize`
before `.ndb`: 16 after `.ndb`: 8

c. `>>> ice.update`
before `.ndb`: ()
after `.ndb`:
`(('N.C', False), ('&', True), ('includeConjuncts', False))`

As the comparison shows,

- 1091 NPs have been removed;
- the maximal pattern size is now 8 (long NPs/patterns are the result of coordination structures);
- the property `.update` indicates the modifications leading to the current db (or rather: current state of the db).

1.4 DB inhabitants – NPs / Patterns / Categories / Lemmata

1.4.1 Brief background: the `Pattern` class

The database inhabitants are noun phrases (NPs) construed as `Pattern` objects; individual NPs can be retrieved via a getter method:

(8) `>>> ice.getNP(index)` with 0 ≤ `index` ≤ (`ice.size - 1`)

a. `>>> np = ice.getNP(1616)`

b. `>>> np`
`<class 'Pattern'>`

c. `>>> print(np)`

```
[Patternize::Pattern; @ID: OIce.083.117;
 patt2:  ('H', 'Md.Aj.Fn', 'N.C', 'GenP')]
```

Via the ID (here: 'OIce.083.117'), the corresponding DB entry can be found in the NPEGL database (including all features, annotations and the respective textual context).

The `Pattern` class itself provides a number of properties and methods to inspect the respective pattern construal(s):

(9) `>>> np.getPatt(level)` with 0 ≤ `level` ≤ 3[1]

(10) a. `>>> np.lemmata`
⇒ `('hinn', 'áttundi', 'dagur', '#')`[2]

b. `>>> ice.getPatt(0)`
⇒ `('H', 'Md', 'N', 'GenP')`

c. `>>> ice.getPatt(1)`
⇒ `('H', 'Md.Aj', 'N.C', 'GenP')`

d. `>>> ice.getPatt(2)`
⇒ `('H', 'Md.Aj.Fn', 'N.C', 'GenP')`

e. `>>> ice.getPatt(3)`
⇒ `('H', 'Md.Aj.Fn.Ord', 'N.C', 'GenP')`

(11) `>>> np.hasCat(cat)`
⇒ returns `True` iff this NP contains `cat`, else `False`

(12) `>>> np.getIndex(cat)`
⇒ returns index position of (the first occurrence of) `cat` iff present, else -1

[1]Methods of the `Patternize` class operating on patterns have the default setting `level=2`.
[2]The hashtag symbol '#' indicates a phrasal category, here 'GenP', that has no lemma proper.

1.4.2 Methods in the `Patternize` class

The `.categorize(level=2)` method can be used to inspect the category inventory at a given level (default setting: 2) of the current working db (returns a `Counter` object):

(13) a. `>>> cats = ice.categorize()`

b. `>>> print(len(cats))` = number of categories
`>>> from pprint import pprint`
`>>> pprint(cats)` = category occurrences:

c.
```
Counter({'N.C': 7981,
         'Md.Aj.Lx':  2013,
         ......
         'PP': 158,
         'CC.Fi':  83,
         ......
        })
```

NB: the number of common nouns ('N.C') is identical to `ice.size` == 7981, which is the effect of calling the `ice.ndb()` method.

Similarly, the `.patternize(level=2)` method returns the pattern inventory (→ `Counter`) at a given level (default setting: 2) of the current working db :

(14) a. `>>> patts = ice.patternize()`

b. `>>> print(len(patts))` = number of patterns
`>>> from pprint import pprint`
`>>> pprint(patts)` = pattern occurrences:

c.
```
Counter({('N.C', 'Poss'):  915,
         ('Md.Aj.Lx', 'N.C'): 807,
         ('N.C', 'GenP'): 653,
         .......
         ('H', 'Md.Aj.Lx', 'N.C', 'GenP'): 8,
         (N.C', 'Dem', 'H', 'Md.Aj.Lx'):  8,
         .......
        })
```

There is also a method that allows to examine, for each individual category, in how many (and which) pattern it occurs:

(15) a. `>>> ice.cat_in_patt(cat, level=2) -> Counter`

b. `>>> relClausePatts = ice.cat_in_patt("RC")`

c. `>>> print(len(relClausePatts))` = number of patterns
`>>> from pprint import pprint`
`>>> pprint(relClausePatts)` = pattern occurrences:

d.
```
Counter({('Dem', 'N.C', 'RC'): 259,
          ('N.C', 'Dem', 'RC'): 194,
          ('Md.Aj.Lx', 'N.C', 'Dem', 'RC'): 60,
          ('Q', 'N.C', 'Dem', 'RC'): 52,
          .......
        })
```

In addition, there are two methods to inspect lemmata:

(16) a. `>>> ice.lemmatize() -> Counter`
⇒ returns the lemmata of all (lexical) categories in the current db

b. `>>> ice.findLemma(cat) -> Counter`
⇒ returns the lemmata instantiating `cat` in the current db

1.5 Pattern Diversity and random databases

Pattern Diversity is a type-token ratio (or proportion), viz.: **patterns per noun phrases**. It is a simple diversity index ($\sim$ mean distribution). A ratio of 1.0 $\leftrightarrow$ 100.0% would indicate maximal diversity – every NP instantiates a different pattern. At the outset, this can be calculated directly in two ways:

(17) a. `>>> patts = ice.patternize()`
`>>> print(len(patts)/ice.size)` $\rightarrow$ 0.073925573236436

b. `>>> ice.patterns_per_NPs()` $\rightarrow$ 7.39%

But as discussed elsewhere (Pfaff 2024, Sect. 2), when comparing dbs of different sizes, this can be misleading; what is required is a *standardized common denominator* (SCD). The procedure provided by *Patternization* is the following:

(i.) create a random subdatabase ($\rightarrow$ initialize the `.rnd_database`, cf. (4)) that has a convenient size with the `.randomize` method:

(18) `>>> ice.randomize(size)` (default setting: `size=1000`)

(19) `>>> print(ice.rnd_database)`
$\Rightarrow$ `[Patternize::Database; Language: Old Icelandic, @rnd_database; Size: 1000]`

(ii.) repeat step (18) n times while adding the number of patterns = p_n found in each `.rnd_database`$_n$

(iii.) divide that sum by n (= average number of patterns) $\Rightarrow$ $\mu = \frac{1}{n}\sum_{i=1}^{n} p_i$

(iv.) divide μ by SCD (= db size = 1000).

As mentioned in Sect. 1.2, the `.rnd_database` attribute is itself a `Patternize` object with all the functionalities mentioned here, notably, the `.patternize` method. This (sub-) database is re-initialized anew every time the `.randomize` method is called. So in principle, the above procedure looks like this:

(20) a. `>>> ice.randomize()`

b. `>>> ice.rnd_update` $\rightarrow$ 1

c. `>>> len(ice.rnd_database.patternize())` $\rightarrow$ 179

(21) a. `>>> ice.randomize()`

b. `>>> ice.rnd_update` $\rightarrow$ 2

c. `>>> len(ice.rnd_database.patternize())` $\rightarrow$ 185

(22) a. `>>> ice.randomize()`

b. `>>> ice.rnd_update` $\rightarrow$ 3

c. `>>> len(ice.rnd_database.patternize())` $\rightarrow$ 201

(23) a. `>>> ice.randomize()`

b. `>>> ice.rnd_update` → 4

c. `>>> len(ice.rnd_database.patternize())` → 178

(24) a. `>>> ice.randomize()`

b. `>>> ice.rnd_update` → 5

c. `>>> len(ice.rnd_database.patternize())` → 193

(25) . . . .

For convenience, the whole procedure is implemented in a single method with the following default settings (`runs` = number of times the procedure is repeated = n):

(26) `.patternDiversity(level=2, runs=500, size=1000)`

(27) a. `>>> ice.patternDiversity` → 18.52

b. `>>> ice.rnd_update` → 505

(28) a. `>>> ice.patternDiversity` → 18.53

b. `>>> ice.rnd_update` → 1005

(29) a. `>>> ice.patternDiversity` → 18.52

b. `>>> ice.rnd_update` → 1505

In a perfect world, the procedure would have to be repeated more often, but given the size of the current db = 7981, we would have to create $\binom{7981}{1000}$ distinct(!) sub-databases, which would result in a somewhat inconvenient runtime behaviour .... . It turns out that 500 repetitions already leads to relatively stable results (ca. $\pm$ 0.1).

NB: via the `(size)` parameter in the `.randomize` method, (randomized) sub-databases can be created of different sizes; alternatively, different subdatabases can be created simultaneously (for various experiments, tests, comparisons etc.):

(30) a. `>>> ice.randomize(3728)`

b. `>>> print(ice.rnd_database)`

```
[Patternize::Database; Language:  Old Icelandic,
 @rnd_database; Size:  3728]
```

(31) a. `>>> ice.randomize()`

b. `>>> rndDB1 = ice.rnd_database`

c. `>>> ice.randomize()`

d. `>>> rndDB2 = ice.rnd_database`

e. `>>> ice.randomize()`

f. `>>> rndDB3 = ice.rnd_database`

1.6 Combinatorial Flexibility

Combinatorial Flexibility is one central component of *Patternization* (see Pfaff 2024, Sect. 3). It creates pattern families, as it were, in the form of permutation groups that are based on some category selection and further specifications. The procedure ⇔ the eponymous method `.combinatorialFlexibility` first creates all → **combinations** of a given length out of a given collection of category labels. IFF the combination contains a certain category label (→ `catCondition`; plausibly, a nominal category; by default: "N.C"), the full series of → **permutations** is generated. Next, the database is browsed and checked whether (and how often) a given permutation (group) is attested. The only mandatory argument to be passed into the method is a collection – `list` or `tuple` – of category labels (→ `cats`). It is possible to use the entire category inventory (discouraged!), or the category labels at a specific level, or a customized collection can be created. In the latter case, the `catCondition` (here: "N.C") must be included. Some categories may occur twice per NP (e.g. adjective); in order to anticipate that eventuality, the respective category should be included twice in the collection, see 'Md.Aj.Lx' in (32b-iii):

(32) a.
```
.combinatorialFlexibility(cats,
                          length=3, count=bool,
                          sm_pattern=None, align=None,
                          pattern_threshold=1, group_threshold=2,
                          catCondition="N.C",
                          .   .   .   .   .
                         )
```

b. i.
```
>>> cats = [cat for cat in ice.getAllCats()]
```

ii.
```
>>> cats = [cat for cat in ice.categorize(i)]
```
with $i \in \{0, 1, 2, 3\}$

iii.
```
>>> cats = ['Md.Aj.Fn', 'Md.Aj.Lx', 'Md.Aj.Lx',
            'Poss', 'Dem', 'Q', 'N.C']
```

c.
```
>>> combis = ice.combinatorialFlexibility(cats)
```

The method returns a collection of `CombFlex` objects; `CombFlex` is a class that allows to store and display information about a given permutation group. The information can conveniently be inspected via the `toString` method:

(33) a.
```
>>> for c in combis:
...       print(c)
```

b.
```
Combination:  {'Poss', 'N.C', 'Md.Aj.Fn'}

   SM_Pattern:  <bound method Patternize.rigid>

   Combinatorial Flexibility:  4 / 6

   Alignment:  None

   GroupCount:  18

   Permutations:  ('Md.Aj.Fn', 'Poss', 'N.C'):   True
                   ('Md.Aj.Fn', 'N.C', 'Poss'):  True
                   ('Poss', 'Md.Aj.Fn', 'N.C'):  True
                   ('Poss', 'N.C', 'Md.Aj.Fn'):  True
                   ('N.C', 'Md.Aj.Fn', 'Poss'):  False
                   ('N.C', 'Poss', 'Md.Aj.Fn'):  False

= = = = = = = = = = = = = = = = = = = = = = = = = = =

Combination:  {'N.C', 'Q', 'Md.Aj.Fn'}

   SM_Pattern:  <bound method Patternize.rigid>

   Combinatorial Flexibility:  5 / 6

   Alignment:  None

   GroupCount:  10

   Permutations:  ('Md.Aj.Fn', 'Q', 'N.C'):   True
                   ('Md.Aj.Fn', 'N.C', 'Q'):  True
                   ('Q', 'Md.Aj.Fn', 'N.C'):  True
                   ('Q', 'N.C', 'Md.Aj.Fn'):  True
                   ('N.C', 'Md.Aj.Fn', 'Q'):  True
                   ('N.C', 'Q', 'Md.Aj.Fn'):  False

= = = = = = = = = = = = = = = = = = = = = = = = = = =
```

. . . . etc.

A given pattern / permutation is considered attested ($\leftrightarrow$ `True`) if it occurs at least `pattern_threshold` times (default: 1). A permutation group (combination) is considered attested if the combined (attested) pattern count $\Leftrightarrow$ `GroupCount` in that group is at least `group_threshold` (default: 2). Only if these two threshold conditions are satisfied will the `.combinatorialFlexibility` method return an output for the respective permutation group.

The parameter `count` is by default set to `bool`, but alternatively, it can be set to `int`; as a result, the actual number of occurrences of the individual permutations will be displayed:

(34) a.
```
>>> combis = ice.combinatorialFlexibility(cats, count=int)
```

b.
```
Combination:  {'Poss', 'Q', 'N.C'}

    SM_Pattern:  <bound method Patternize.rigid>

    Combinatorial Flexibility:  5 / 6

    Alignment:  None

    GroupCount:  138

    Permutations:   ('Poss', 'Q', 'N.C'):  1
                    ('Poss', 'N.C', 'Q'):  1
                    ('Q', 'Poss', 'N.C'):  40
                    ('Q', 'N.C', 'Poss'):  81
                    ('N.C', 'Poss', 'Q'):  15
                    ('N.C', 'Q', 'Poss'):  0
```

`SM_Pattern:  <bound method Patternize.   >` indicates the search filter used for the query (see Pfaff 2024, Sect. 4). There are three possible values for the parameter `sm_pattern` (= Search/Match Pattern): `.rigid`; `.precise`; `.flexi`; those are functional parameters implemented as instance methods (and have to be called via instance name; default: `.rigid`):

(35) a.
```
>>> combis = ice.combinatorialFlexibility(cats, count=int,
                                              sm_pattern=ice.flexi)
```

b.
```
Combination:  {'Poss', 'Q', 'N.C'}

    SM_Pattern:  <bound method Patternize.flexi>

    Combinatorial Flexibility:  5 / 6

    Alignment:  None

    GroupCount:  150

    Permutations:   ('Poss', 'Q', 'N.C'):  1
                    ('Poss', 'N.C', 'Q'):  1
                    ('Q', 'Poss', 'N.C'):  42
                    ('Q', 'N.C', 'Poss'):  90
                    ('N.C', 'Poss', 'Q'):  16
                    ('N.C', 'Q', 'Poss'):  0
```

1.7 Schrödinger's CATs

1.7.1 The Prenominal/Postnominal Domain

see Pfaff (2024, Sect. 5)

(36) a.
```
>>> from pprint import pprint
```

b.
```
>> print(ice.prenominalDomain)
```

c.
```
>> print(ice.postnominalDomain)
```

d.
```
>> ice.partitionize()  =  ice.partitionize('pre')
```

e.
```
>> ice.partitionize('post')
```

f.
```
>> print(ice.prenominalDomain)
```

g.
```
>> print(ice.postnominalDomain)
```

h.
```
>> pre = ice.prenominalDomain
```

i.
```
>> post = ice.postnominalDomain
```

j.
```
>> pre.patternize()
```

k.
```
>> post.cat_in_patt('Dem')
```

(37) a.
```
>>> cats = ["Dem","Md.Aj","Md.Aj.Lx","Md.Aj.Lx",
             "Md.Aj.Fn","Md.Aj.Fn","Poss", "Q", "H"]
```

b.
```
>>> combis = pre.combinatorialFlexibility(cats, catCondition="Md.Aj")
```

c.
```
>>> for c in combis:
        print(c)
```

. . . . etc.

1.7.2 Distance from N

(38) a.
```
>>> pprint(pre.probabilize())
```

b.
```
>>> pprint(post.probabilize())
```

c.
```
>>> pprint(ice.probabilize())
```
(for comparison)

d. `>>> pprint(ice.getPartitions(partition='pre'))`
e. `>>> pprint(ice.getPartitions(partition='post'))`
f. `>>> pprint(ice.getPartitions(countFrom='right'))`

g. `>>> pprint(pre.pair_wize(order='precede'))`
h. `>>> pprint(pre.pair_wize(order='follow'))`
i. `>>> pprint(post.pair_wize(order='precede'))`
j. `>>> pprint(post.pair_wize(order='follow'))`
k. `>>> pprint(pre.pair_wize(distance=2))`

1.7.3 Visualize!

NB: experimental stage; not actually implemented yet !!!

(39) a. `>>> pre.showPobabilisticCatDistribution()`
b. `>>> post.showPobabilisticCatDistribution()`
c. `>>> ice.showPobabilisticCatDistribution()`

`>>> . . . to be continued!`

References

Pfaff, Alexander. 2019. NPEGL – Annotation manual. Ms., University of Oslo.

Pfaff, Alexander. 2024. How to measure syntactic diversity: Patternization – methods and algorithms. In *Noun phrases in early Germanic languages*, eds. Kristin Bech and Alexander Pfaff, 33–70. Language Science Press.

Pfaff, Alexander, and Gerlof Bouma. 2024. The NPEGL noun phrase database: design and construction. In *Noun phrases in early Germanic languages*, eds. Kristin Bech and Alexander Pfaff, 1–32. Language Science Press.