pygamma-agreement

Rachid RIAD, Hadrien TITEUX, Léopold FAVRE

Apr 14, 2023

CONTENT:

pygamma-agreement is a Python library used to measure the inter-annotator agreement, defined by Mathet et Al. in "The Unified and Holistic Method Gamma for Inter-Annotator Agreement Measure and Alignment" . It features an easy-to-use API and a a command line interface (CLI) for those that prefer using regular shell scripts for their data processing tasks.

CHAPTER

ONE

INSTALLATION

The package is available on pip. Just run

```
$ pip install pygamma-agreement
```

Pygamma-agreement uses the GNU Linear Programming Kit as its default Mixed Integer Programming solver (critical step of the gamma-agreement algorithm). Since it is quite slow, you can install the CBC solver and its official python API. To use those in *pygamma-agreement*, simply install them:

- **Ubuntu/Debian** : `$ sudo apt install coinor-libcbc-dev`
- **Fedora** : `$ sudo yum install coin-or-Cbc-devel`
- **Arch Linux** : `$ sudo pacman -S coin-or-cbc`
- **Mac OS X :**
  - `$ brew tap coin-or-tools/coinor`
  - `$ brew install coin-or-tools/coinor/cbc pkg-config`

then:

```
$ pip install "pygamma-agreement[CBC]"
```

Warning: A bug in GLPK causes the standart ouput to be polluted by non-deactivable messages. If you expect to use standart output for informative or parsing purposes, it is strongly advised to use the CBC solver.

1.1 Quickstart

In this small tutorial, you'll learn how to use the essential features of pygamma-agreement.

1.1.1 Loading the data

Let's say we have a short recording of some human speech, in which several people are chatting. Let's call these people **Robin**, **Maureen** and **Marvin**. We asked 3 annotators to annotate this audio file to indicate when each of these people is talking. Each annotated segment can be represented as a tuple of 3 information:

- Who is speaking ("Marvin")
- Segment start (at 3.5s)
- Segment end (at 7.2s)

Obviously, our annotators sometimes disagree on who might be talking, or when exactly each person's speech turn is starting or ending. Luckily, the Gamma inter-annotator agreement enables us to measure that.

We'll first load the annotation into `pygamma-agreement`'s base data structure, the `Continuum`, made to store this kind of annotated data.

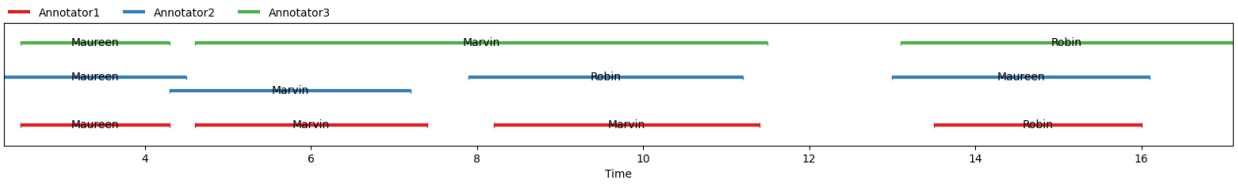
```
from pygamma_agreement import Continuum
from pyannote.core import Segment

continuum = Continuum()
continuum.add("Annotator1", Segment(2.5, 4.3), "Maureen")
continuum.add("Annotator1", Segment(4.6, 7.4), "Marvin")
continuum.add("Annotator1", Segment(8.2, 11.4), "Marvin")
continuum.add("Annotator1", Segment(13.5, 16.0), "Robin")

continuum.add("Annotator2", Segment(2.3, 4.5), "Maureen")
continuum.add("Annotator2", Segment(4.3, 7.2), "Marvin")
continuum.add("Annotator2", Segment(7.9, 11.2), "Robin")
continuum.add("Annotator2", Segment(13.0, 16.1), "Maureen")

continuum.add("Annotator3", Segment(2.5, 4.3), "Maureen")
continuum.add("Annotator3", Segment(4.6, 11.5), "Marvin")
continuum.add("Annotator3", Segment(13.1, 17.1), "Robin")
```

If you were to show the `continuum` variable in an jupyter notebook, this is what would be displayed:



The same image can also be displayed with `matplotlib` by using :

```
from pygamma_agreement import show_continuum
show_continuum(continuum, labelled=True)
```

1.1.2 Setting up a dissimilarity

To measure our inter-annotator agreement (or disagreement), we'll need a dissimilarity. Dissimilarities can be understood to be like distances, although they don't quite satisfy some of their theoretical requirements.

In our case, we want that dissimilarity to measure both the disagreement on **segment boundaries** (when is someone talking?) and **segment annotations** (who's talking?). Thus, we'll be using the `CombinedCategoricalDissimilarity`, which uses both **temporal** and **categorical** data to measure the disagreement between annotations.

Since we think that eventual categorical mismatches are more important than temporal mismatches, we'll assign a greater weight to the former using the `alpha` (for temporal mismatches) and `beta` (for categorical mismatches) coefficients.

```
from pygamma_agreement import CombinedCategoricalDissimilarity

dissim = CombinedCategoricalDissimilarity(alpha=1, beta=2)
```

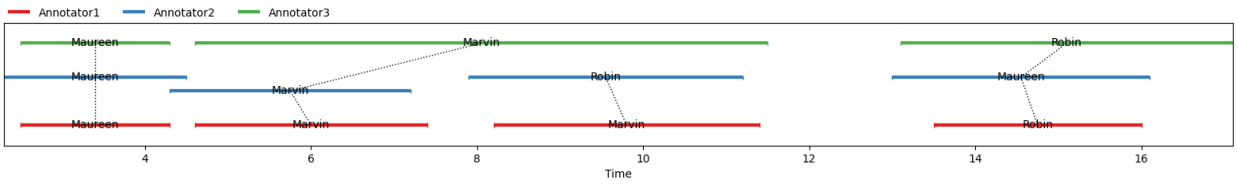
1.1.3 Computing the Gamma

We're all set to compute the gamma agreement now!

```
gamma_results = continuum.compute_gamma(dissim)
print(f"The gamma for that annotation is f{gamma_results.gamma}")
```

As a side note: the computation of the gamma value requires that we find the alignment with lowest possible disorder (this being condition by the dissimilarity that was chosen beforehand). We can easily retrieve that alignment, and display it (if we're in a jupyter notebook):

```
gamma_results.best_alignment
```



The same image can also be displayed with `matplotlib` by using :

```
from pygamma_agreement import show_alignment
show_alignment(gamma_results.best_alignment, labelled=True)
```

1.1.4 The whole example

Here's the full quickstart code example, for convenience:

```
from pyannote.core import Segment

from pygamma_agreement import (CombinedCategoricalDissimilarity,
                               Continuum,
                               show_alignment,
                               show_continuum)
```

(continues on next page)

(continued from previous page)

```
continuum = Continuum()
continuum.add("Annotator1", Segment(2.5, 4.3), "Maureen")
continuum.add("Annotator1", Segment(4.6, 7.4), "Marvin")
continuum.add("Annotator1", Segment(8.2, 11.4), "Marvin")
continuum.add("Annotator1", Segment(13.5, 16.0), "Robin")

continuum.add("Annotator2", Segment(2.3, 4.5), "Maureen")
continuum.add("Annotator2", Segment(4.3, 7.2), "Marvin")
continuum.add("Annotator2", Segment(7.9, 11.2), "Robin")
continuum.add("Annotator2", Segment(13.0, 16.1), "Maureen")

continuum.add("Annotator3", Segment(2.5, 4.3), "Maureen")
continuum.add("Annotator3", Segment(4.6, 11.5), "Marvin")
continuum.add("Annotator3", Segment(13.1, 17.1), "Robin")

dissim = CombinedCategoricalDissimilarity(alpha=1, beta=2)

gamma_results = continuum.compute_gamma(dissim)

print(f"The gamma for that annotation is f{gamma_results.gamma}")
```

1.2 Principles

pygamma-agreement provides a set of classes that can be used to compute the -agreement measure on different sets of annotation data and in different ways. What follows is a detailed explanation of how these classes can be built and used together.

Warning: A great part of this page is just a rehash of concepts that are explained in a much clearer and more detailed way in the original -agreement paper [mathet2015]. This documentation is mainly aimed at giving the reader a better understanding of our API's core principles. If you want a deeper understanding of the gamma agreement, we strongly advise that you take a look at this paper.

1.2.1 Units

A **unit** is an object that is defined in time (meaning, it has a starting point and an ending point), and might bear an annotation. In our case, the only supported type of annotation is a category. In *pygamma-agreement*, for all basic usages, `Unit` objects are built automatically when you add an annotation to *a continuum* . You might however have to create `Unit` objects if you want to create your own *Alignments, Unitary Alignments* .

Note: *Unit* instances are sortable, meaning that they satisfy a total ordering. The ordering uses their temporal positions (starting/ending times) as a first parameter to order them, and if these are strictly equal, uses the alphabetical order to compare them. Ergo,

```
>>> Unit(Segment(0,1), "C") < Unit(Segment(2,3), "A")
True
>>> Unit(Segment(0,1), "C") > Unit(Segment(0,1), "A")
True
```

(continues on next page)

(continued from previous page)

```
>>> Unit(Segment(0,1)) < Unit(Segment(0,1), "A")
True
```

1.2.2 Continua (or Continuums)

A **continuum** is an object that that stores the set of annotations produced by several annotators, all referring to the same annotated file. It is equivalent to the term *Annotated Set* used in Mathet et Al. [mathet2015]. Annotations are stored as *Units* , each annotator's units being stored in a sorted set.

Continuums are the "center piece" of our API, from which most of the work needed to obtain gamma-agreement measures can be done. From a `Continuum` instance, it's possible to compute and obtain:

- its *best alignment* .
- its *gamma agreement measure* .

You can create a continuum then add each unit one by one, by specifying the annotators of the units:

```
continnum = Continuum()

continuum.add('annotator_a',
              Segment(10.5, 15.2),
              'category_1')
```

Continua can also be imported from CSV files with the following structure :

```
annotator, annotation, segment_start, segment_end
```

Thus, for instance:

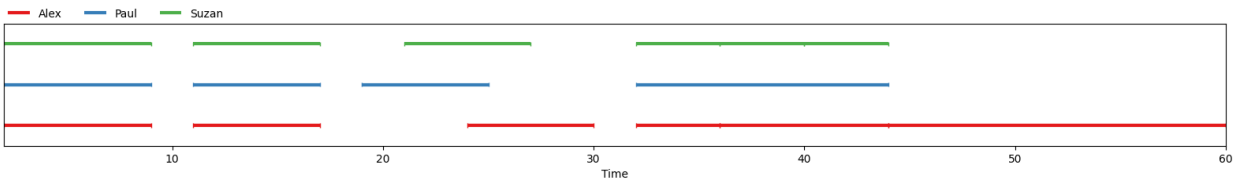
```
E.G.:

annotator_1, Marvin, 11.3, 15.6
annotator_1, Maureen, 20, 25.7
annotator_2, Marvin, 10, 26.3
annotator_C, Marvin, 12.3, 14
```

```
continuum = Continuum.from_csv('your/continuum/file.csv')
```

If you're working in a Jupyter Notebook, outputting a *Continuum* instance will automatically show you a graphical representation of that *Continuum*:

```
In  [mathet2015]_: continuum = Continuum.from_csv("data/PaulAlexSuzan.csv")
                   continuum
```



The same image can also be displayed with `matplotlib` by using :

```
from pygamma_agreement import show_continuum
show_continuum(continuum, labelled=True)
```

1.2.3 Alignments, Unitary Alignments

A unitary alignment is a tuple of *units*, each belonging to a unique annotator. For a given *continuum* containing annotations from `n` different annotators, the tuple will *have* to be of length `n`. It represents a "match" (or the hypothesis of an agreement) between units of different annotators. A unitary alignment can contain *empty units*, which are "fake" (and null) annotations inserted to represent the absence of corresponding annotation for one or more annotators.

An alignment is a set of *unitary alignments* that constitute a *partition* of a continuum. This means that

- each and every unit of each annotator from the partitioned continuum can be found in the alignment
- each unit can be found once and *only* once

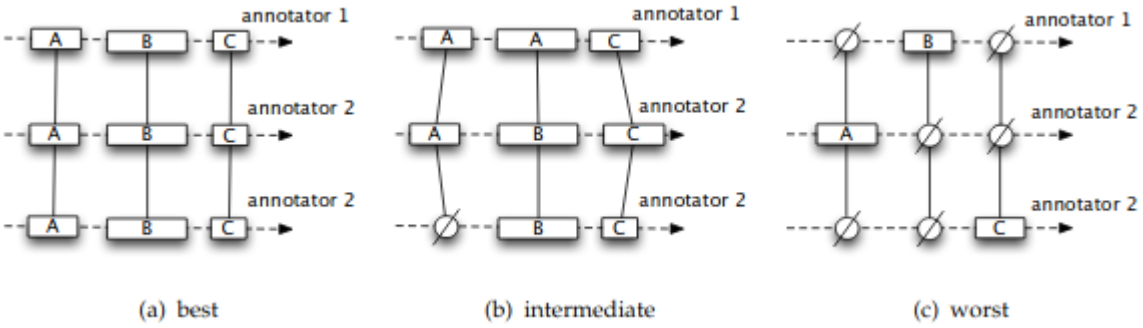


Fig. 1: Visual representations of unitary alignments taken from [mathet2015] , with varying disorders.

For both alignments and unitary alignments, it is possible to compute a corresponding *disorder*. This possible via the `compute_disorder` method, which takes a *dissimilarity* as an argument. This (roughly) corresponds to the disagreement between annotators for a given alignment or unitary alignment.

In practice, you shouldn't have to build both unitary alignments and alignments yourself, as they are automatically constructed by a *continuum* instance's `get_best_alignment` method.

Best Alignments

For a given *continuum*, the alignment with the lowest possible disorder. This alignment is found using a combination of 3 steps:

- the computation of the disorder for all potential unitary alignments
- a simple heuristic explained in [mathet2015]'s section 5.1.1 to eliminate a big part of those potential unitary alignments based on their disorder
- the usage of multiple optimization problem formulated as Mixed-Integer Programming (MIP)

The function that implements these 3 steps is, by far, the most compute-intensive in the whole `pygamma-agreement` library. The best alignment's disorder is used to compute the *The Gamma () agreement* .

1.2.4 Disorders

The disorder for either an *alignment or a unitary alignment* corresponds to the disagreement between its constituting units.

- the disorder between two units is directly computed using a dissimilarity.
- the disorder between units of a unitary alignment is an average of the disorder of each of its unit couples
- the disorder of an alignment is the mean of its constituting unitary alignments's disorders

What should be remembered is that a *dissimilarity* is needed to compute the disorder of either an alignment or a unitary alignment, and that its "raw" value alone isn't of much use but it is needed to compute the -agreement.

1.2.5 Dissimilarities

A dissimilarity is a function that "tells to what degree two units should be considered as different, taking into account such features as their positions, their annotations, or a combination of the two." [mathet2015]. A dissimilarity has the following mathematical properties:

- it is positive ($dissim(u, v) > 0$))
- it is symmetric ($dissim(u, v) = dissim(v, u)$)
- $dissim(x, x) = 0$

Although dissimilarities do look a lot like distances (in the mathematical sense), they don't necessarily are distances. Hence, they don't necessarily honor the triangular inequality property.

If one of the units in the dissimilarity is the *empty unit*, its value is $\Delta_\emptyset$. This value is constant, and can be set as a parameter of a dissimilarity object before it is used to compute an alignment's disorder.

Right now, there are three types of dissimilarities available :

- The positionnal sporadric dissimilarity (we will call it simply Positionnal dissimilarity)
- The categorical dissimilarity
- The combined dissimilarity

Note: Although you will have to instantiate dissimilarities objects when using `pygamma-agreement`, you'll never have to use them in any way other than just by passing it as an argument as shown beforehand.

Positional Dissimilarity

A positional dissimilarity is used to measure the *positional* or *temporal* disagreement between two annotated units u and v.

[mathet2015] introduces the positional sporadic dissimilarity as the reference positional dissimilarity for the gamma-agreement. Its formula is :

$$d_{pos}(u, v) = \left(\frac{|start(u) - start(v)| + |end(u) - end(v)|}{(duration(u) + duration(v))} \right)^2 \cdot \Delta_\emptyset$$

Here's how to instanciate a `PositionnalSporadicDissimilarity` object :

```
from pygamma_agreement import PositionalDissimilarity
dissim_pos = PositionalSporadicDissimilarity(delta_empty=1.0)
```

and that's it. You can also use `dissim_pos.d(unit1, unit2)` to obtain directly the value of the dissimilarity between two units. It is however very unlikely that one would need to use it, as this value is only needed in the very low end of the gamma computation algorithm.

Categorical Dissimilarity

A categorical dissimilarity is used to measure the *categorical* disagreement between two annotated units u and v.

$$d_{cat}(u, v) = dist_{cat}(cat(u), cat(v)) \cdot \Delta_{\emptyset}$$

In our case, the function $dist_{cat}$ is computed using a simple lookup in a *categorical distance matrix* `D`. Let's suppose we have K categories, this matrix will be of shape `(K,K)`.

Here is an example of a distance matrix for 3 categories:

```
>>> D
array([[0. , 0.5, 1. ],
       [0.5, 0. , 1. ],
       [1. , 1. , 0. ]])
```

To comply with the properties of a dissimilarity, the matrix has to be symmetrical, and has to have an empty diagonal. Moreover, its values have be between 0 and 1. By default, for two units with differing categories, $d_{cat}(u, v) = 1$, and thus the corresponding matrix is:

```
>>> D_default
array([[0., 1., 1. ],
       [1., 0., 1. ],
       [1., 1., 0. ]])
```

here's how to instanciate a categorical dissimilarity:

```
D = array([[ 0.,   0.5,    1. ],
           [0.5,    0.,  0.75 ],
           [ 1.,  0.75,    0. ]])

from pygamma_agreement import PrecomputedCategoricalDissimilarity
from sortedcontainers import SortedSet

categories = SortedSet(('Noun', 'Verb', 'Adj'))
dissim_cat = PrecomputedCategoricalDissimilarity(categories,
                                                 matrix=D,
                                                 delta_empty=1.0)
```

Warning: It's important to note that the index of each category in the categorical dissimilarity matrix is its index in **alphabetical order**. In this example, the considered dissimilarity will be:

- $dist_{cat}('Adj','Noun') = 0.5$
- $dist_{cat}('Adj','Verb') = 1.0$
- $dist_{cat}('Noun','Verb') = 0.75$

You can also use the default categorical dissimilarity, the `AbsoluteCategoricalDissimilarity`; there are also other available categorical dissimilarities, such as the `LevenshteinCategoricalDissimilarity` which is based on the levenshtein distance.

Combined Dissimilarity

The combined categorical dissimilarity uses a linear combination of the two previous *categorical* and *positional* dissimilarities. The two coefficients used to weight the importance of each dissimilarity are α and β :

$$d_{combi}^{\alpha,\beta}(u,v) = \alpha.d_{pos}(u,v) + \beta.d_{cat}(u,v)$$

This is the dissimilarity recommended by [mathet2015] for computing gamma.

It takes the same parameters as the two other dissimilarities, plus α and β :

```
from pygamma_agreement import CombinedCategoricalDissimilarity,␣
↪LevenshteinCategoricalDissimilarity

categories = ['Noun', 'Verb', 'Adj']
dissim = CombinedCategoricalDissimilarity(alpha=3,
                                          beta=1,
                                          delta_empty=1.0,
                                          pos_dissim=PositionalSporadicDissimilarity()
                                          cat_
↪dissim=LevenshteinCategoricalDissimilarity(categories))
```

1.2.6 The Gamma () agreement

The -agreement is a *chance-adjusted* measure of the agreement between annotators. To be computed, it requires

- a *continuum* , containing the annotators's annotated units.
- a *dissimilarity*, to evaluate the disorder between the hypothesized alignments of the annotated units.

Using these two components, we can compute the *best alignment* and its disorder. Let's call this disorder δ_{best}.

Without getting into its details, our package implements a method of sampling random annotations from a continuum. Using these N sampled continuum, we can also compute a best alignment and its subsequent disorder. Let's call these disorders δ_{random}^{i}, and the mean of these values $\delta_{random} = \frac{\sum_i \delta_{random}^{i}}{N}$

The gamma agreement's formula is finally:

$$\gamma = 1 - \frac{\delta_{best}}{\delta_{random}}$$

Several points that should be clarified about that value:

- it is bounded by $]-\infty, 1]$ but for most "regular" situations it should be contained within $[0, 1]$
- the higher and the closer it is to 1, the more similar the annotators' annotations are.

the gamma value is computed from a `Continuum` object, using a given `Dissimilarity` object :

```
continuum = Continuum.from_csv('your/csv/file.csv')
dissim = CombinedCategoricalDissimilarity(delta_empty=1,
                                          alpha=3,
                                          beta=1)
gamma_results = continuum.compute_gamma(dissim,
                                        precision_level=0.02)

print(f"gamma value is: {gamma_results.gamma}")
```

Warning: The algorithm implemented by `continuum.compute_gamma` is very costly. An approximation of its computational complexity would be $O(N \times (p_1 \times ... \times p_n))$ where p_i is the number of annotations for annotator i, and N is the number of samples used when computing δ_{random}, which grows as the `precision_level` parameter gets closer to 0. If time of computation becomes too high, it is advised to lower the precision before anything else.

1.2.7 Gamma-cat (-cat) and Gamma-k (-k)

$_{cat}$ is an alternate inter-annotator agreement measure based on , made to evaluate the task of categorizing pre-defined units. Just like , it is a *chance-adjusted* metric :

$$\gamma_{cat} = 1 - \frac{\delta^{cat}_{best}}{\delta^{cat}_{random}}$$

Where the disorder δ_{cat} of a continuum is computed using the same *best alignment* used for the -agreement : this disorder is basically the average dissimilarity between pairs of non-empty *units* in every *unitary alignment*, weighted by the possitional agreement between the units.

The $_{cat}$-agreement can be obtained from the *GammaResults object* easily:

```
print(f"gamma-cat value is : {gamma_results.gamma_cat} ")
```

$_{k}$ is another alternate agreement measure. It only differs from $_{cat}$ by the fact that it only considers one defined category.

The $_{k}$ value for a category alse can be obtained from the *GammaResults object*:

```
for category in continuum.categories:
    print(f"gamma-k of '{category}' is : {gamma_results.gamma_k(category)} ")
```

Further details about the measures and the algorithms used for computing them can be consulted in [mathet2015].

On a side-note, we would recomment combining gamma-cat with the alternate inter-annotator user agreement measure **Soft-Gamma** (new measure whose idea emerged during the developpement of `pygamma-agreement`). More information about soft-gamma is available in the dedicated section of this documentation.

1.3 Command Line Tool

The `pygamma-agreement` CLI enables you to compute the gamma inter-annotator agreement without having to use our library's Python API. For now, and to keep its usage as simple as possible, it only supports the Combined Categorical Dissimilarity.

To use the command line tool, make sure you've installed the `pygamma-agreement` package via pip. You can check that the CLI tool is properly installed in your environment by running

```
pygamma-agreement -h
```

1.3.1 Supported data formats

Pygamma-agreement's CLI takes CSV or RTTM files as input. CSV files need to have the following structure:

```
annotator, annotation, segment_start, segment_end
```

Thus, for instance:

```
annotator_1, Marvin, 11.3, 15.6
annotator_1, Maureen, 20, 25.7
annotator_2, Marvin, 10, 26.3
annotator_C, Marvin, 12.3, 14
```

1.3.2 Using the command line tool

`pygamma-agreement`'s command line can be used on one or more files, or a folder containing several files. Thus, all these commands will work:

```
pygamma-agreement data/my_annotation_file.csv
pygamma-agreement data/annotation_1.csv data/annotation_2.csv
pygamma-agreement data/*.csv
pygamma-agreement data/
```

The gamma value for each file will be printed out in the console's stdout. If you wish to easily save the tool's output in a file, you can use the `--output-csv` (or `-o`) option to save it to a CSV that will contain each file's gamma value. You can then visualize the results with tools like `tabview` or other spreadsheet software alike.

```
pygamma-agreement data/*.csv --output-csv gamma_results.csv
pygamma-agreement data/*.csv -o gamma_results.csv
```

There is also the option of saving the values in the JSON format with the `--output-json` (or `-j`) option.

```
pygamma-agreement data/*.csv --output-json gamma_results.json
pygamma-agreement data/*.csv -j gamma_results.json
```

Optionally, you can also configure the alpha and beta coefficients used in the combined categorical dissimilarity. You can also set a different precision level if you want your gamma's estimation to be more or less precise:

```
pygamma-agreement data/my_annotation_file.csv --alpha 3 --beta 1 --precision-level 0.02
```

In addition the gamma agreement, `pygamma-agreement` also gives you the option to output the gamma-cat & gamma-k(s) alternate inter-annotator agreements, with the `--gamma-cat` (or `-g`) and `--gamma-k` (or `-k`) options, or both.

```
pygamma-agreement -k -g tests/data/*.csv -o test.csv
OR
pygamma-agreement -k -g tests/data/*.csv -j test.csv
```

1.4 How-to's

This section will explain how to make the most of `pygamma-agreement`'s customization possibilities.

1.4.1 Setting up your own positional dissimilarity

The only positionnal dissimilarity available in `pygamma-agreement` is the *Positional Sporadic dissimilarity*, introduced by [mathet2015]. In this part, we'll detail how to use your own for computing the gamma agreement.

A `Dissimilarity` class is a class that holds the information needed to **compile** a dissimilarity function. It is made this way to optimize as much as possible the very costly computation of gamma. Let's start by **inheriting** the `AbstractDissimilarity` class :

```
from pygamma_agreement import AbstractDissimilarity, Unit
import numpy as np
from typing import Callable

class MyPositionalDissimilarity(AbstractDissimilarity):
    def __init__(self, delta_empty=1.0):
        super().__init__(delta_empty=delta_empty)

    # Abstract methods overrides
    def compile_d_mat(self) -> Callable[[np.ndarray, np.ndarray], float]:
        ...

    def d(self, unit1: Unit, unit2: Unit) -> float:
        ...
```

This is essentially the minimum skeleton needed to define a new positional dissimilarity. You can also use additionnal attributes to make your dissimilarity more parametrizable.

For this example, the dissimilarity we'll be implementing will be defined as such : for an integer p,

$$d_p(u, v) = (|start(u) - start(v)|^p + |end(u) - end(v)|^p)^{\frac{1}{p}} \times \Delta_\emptyset$$

$$d_p(u_\emptyset, u) = d_p(u, u_\emptyset) = \Delta_\emptyset$$

thus, let's redefine the constructor accordingly :

```
def __init__(self, p: int, delta_empty=1.0):
        self.p = p
        assert p > 0
        super().__init__(delta_empty=delta_empty)
```

Great. Now, let's explain what the `d` method is.

The `AbstractDissimilarity.d` method is essentially the python method that returns the dissimilarity between the two non-empty given units. When the units are empty, any algorithm from `pygamma-agreement` that uses a dissimilarity object will do the checking out of this method and use the `delta_empty` attribute accordingly.

So, you only have to implement the first part of d_p's definition.

```
def d(self, unit1: Unit, unit2: Unit) -> float:
    return ( abs(unit1.segment.start - unit2.segment.start)**self.p
             + abs(unit1.segment.end - unit2.segment.end)**self.p ) ** (1/self.p)
```

This method is essentially used in the gamma-cat computation, since it is too slow for the costly gamma algorithm.

It's time to explain the essential part of the dissimilarity : the `compile_d_mat` method. To minimize computation time of the gamma-agreement, the dissimilarities are used in a C-compiled form generated by the `numba` library. To accomplish this, they need to keep the same signature, and to only use native python or `numpy` types/operations internally.

Let's write the `MyPositionalDissimilarity.compile_d_mat` method to better explain it :

```
def compile_d_mat(self) -> Callable[[np.ndarray, np.ndarray], float]:
    # Calling self inside d_mat makes the compiler choke, so you need to copy attributes␣
↪in locals.
    p = self.p
    delta_empty = self.delta_empty
    from pygamma_agreement import dissimilarity_dec

    @dissimilarity_dec  # This decorator specifies that this function will be compiled.
    def d_mat(unit1: np.ndarray, unit2: np.ndarray) -> float:
        # We're in numba environment here, which means that only python/numpy types and␣
↪operations will work.
        return (abs(unit1[0] - unit2[0])**p
                + abs(unit1[1] - unit2[1])**p)**(1 / p) * delta_empty

    return d_mat
```

You'll notice that the units' attributes are accessed by index. The correspondance is the following :

```
unit_array: np.ndarray
unit_object: Unit

unit_array[0] == unit_object.segment.start
unit_array[1] == unit_object.segment.end
unit_array[2] == unit_object.segment.end - unit_object.segment.start
```

Now, the dissimilarity is ready to be used !

```
from pygamma_agreement import Continuum
continuum: Continuum
dissim = MyPositionalDissimilarity(p=2, delta_empty=1.0)
gamma_results = continuum.compute_gamma(dissim)
```

Warning: A very important thing to note is that the structure of dissimilarities is not really compatible with changing attributes, because of the class structure and of compilation. It is advised do **redefine** your dissimilarities if you want to change attributes.

```
dissim.p = 3 # DON'T do that !
dissim = MyPositionalDissimilarity(p=3, delta_empty=1.0) # Redefine it instead.
```

1.4.2 Setting up your own categorical dissimilarity

For many reasons, string types are not easy to manipulate in `numba njit`'ed code. Instead, category-to-category dissimilarities are pre-computed at the python level. Thus, there is a very simple interface avaible : You just need to inherit the `LambdaCategoricalDissimilarity`, and override the `cat_dissim_func` static method :

```
class MyCategoricalDissimilarity(LambdaCategoricalDissimilarity):
# Precomputation requires the category labels to be saved. Don't use this dissimilarity␣
↪with
# a continuum containing unspecified categories
def __init__(self, labels: Iterable[str], delta_empty: float = 1.0):
    super().__init__(labels, delta_empty)

@staticmethod
def cat_dissim_func(str1: str, str2: str) -> float:
    return ...  # Your categorical dissimilarity function. Results should be in [0, 1]
```

Beware that in reality, the resulting dissimilarity between categories `a` and `b` will be `cat_dissim_func(a, b) * delta_empty`

Your new categorical dissimilarity is now ready. You can, for instance, use it in a combined categorical dissimilarity :

```
from pygamma_agreement import CombinedCategoricalDissimilarity, Continuum

continuum: Continuum
dissim = CombinedCategoricalDissimilarity(alpha=3, beta=1,
                                          cat_
↪dissim=MyCategoricalDissimilarity(continuum.categories))
gamma_results = continuum.compute_gamma(dissim)
```

1.4.3 Combining dissimilarities

The only combined dissimilarity (a dissmilarity that considers both positionning and categorizing of units) natively available in `pygamma-agreement` is the one introduced by [mathet2015] (the `CombinedCategoricalDissimilarity`):

$$d_{\alpha,\beta}(u, v) = \alpha d_{pos}(u, v) + \beta d_{cat}(u, v)$$

$$d_{\alpha,\beta}(u_\emptyset, u) = d_{\alpha,\beta}(u, u_\emptyset) = \Delta_\emptyset$$

Imagine instead that you want to adapt this this dissimilarity geometrically :

$$d_{\alpha,\beta}(u, v) = d_{pos}(u, v)^\alpha \times d_{cat}(u, v)^\beta$$

$$d_{\alpha,\beta}(u_\emptyset, u) = d_{\alpha,\beta}(u, u_\emptyset) = \Delta_\emptyset$$

Let's start by using the same skeleton as for a simple positional dissimilarity :

```
from pygamma_agreement import AbstractDissimilarity, Unit
import numpy as np
from typing import Callable

class MyCombinedDissimilarity(AbstractDissimilarity):
    def __init__(self, alpha: float, beta: float,
```

(continues on next page)

(continued from previous page)

```
                pos_dissim: AbstractDissimilarity,
                cat_dissim: CategoricalDissimilarity,
                delta_empty=1.0):
        self.alpha, self.beta = alpha, beta
        self.pos_dissim, self.cat_dissim = pos_dissim, cat_dissim
        super().__init__(cat_dissim.categories, delta_empty=delta_empty)

    # Abstract methods overrides
    def compile_d_mat(self) -> Callable[[np.ndarray, np.ndarray], float]:
        ...

    def d(self, unit1: Unit, unit2: Unit) -> float:
        ...
```

Note: One important thing to note that if your dissimilarity takes categories into account, you **must** specify a set of categories to the super constructor. Here in this example, the cat_dissim part does take it into account, so its categories can be obtained directly.

The d method can simply make use of the other dissimilarities' d s :

```
def d(self, unit1: Unit, unit2: Unit) -> float:
        return ( self.pos_dissim.d(unit1, unit2)**(self.alpha)
                * self.cat_dissim.d(unit1, unit2)**(self.beta) )
```

Moreover, you can access a dissimilarity's numba-compiled function from the d_mat attribute, which is usable in numba-compiled environment. Thus, compiling the dissimilarity function is pretty simple, and very similar to a simple dissimilarity with only arithmetic operations. Let's illustrate this :

```
def compile_d_mat(self) -> Callable[[np.ndarray, np.ndarray], float]:
        alpha, beta = self.alpha, self.beta
        pos, cat = self.pos_dissim.d_mat, self.cat_dissim.d_mat
        # d_mat attribute contains the numba-compiled function

        from pygamma_agreement import dissimilarity_dec
        @dissimilarity_dec
        def d_mat(unit1: np.ndarray, unit2: np.ndarray) -> float:
            return pos(unit1, unit2)**alpha * cat(unit1, unit2)**beta

        return d_mat
```

Then, there's only the d method left to code.

```
def d(self, unit1: Unit, unit2: Unit):
    return (self.pos_dissim.d(unit1, unit2)**self.alpha *
            self.cat_dissim.d(unit1, unit2)**self.beta)
```

And that's it ! Now, in theory, you have all the tools you need to compute the gamma-agreement with any dissimilarity.

```
continuum: Continuum
dissim = MyCombinedDissimilarity(alpha=3, beta=2,
                                 pos_dissim=MyPositionalDissimilarity(),
```

(continues on next page)

(continued from previous page)

```
                                    cat_dissim=MyCategoricalDissimilarity(continuum.
↪categories))
gamma_results = continuum.compute_gamma(dissim)
```

1.4.4 Generating random continua for comparison using the Statistical Sampler and the Corpus Shuffling Tool

If you want to measure your dissimilarity's influence on the gamma-agreement depending on certain possible errors between annotations, `pygamma-agreement` contains all the the needed tools : the `StatisticalContinuumSampler`, which generates totally random continuua, and the `CorpusShufflingTool`, that simulates errors when annotating a resource.

First, let's generate a random reference for the corpus shuffling tool, which will act as the perfectly accurate annotation:

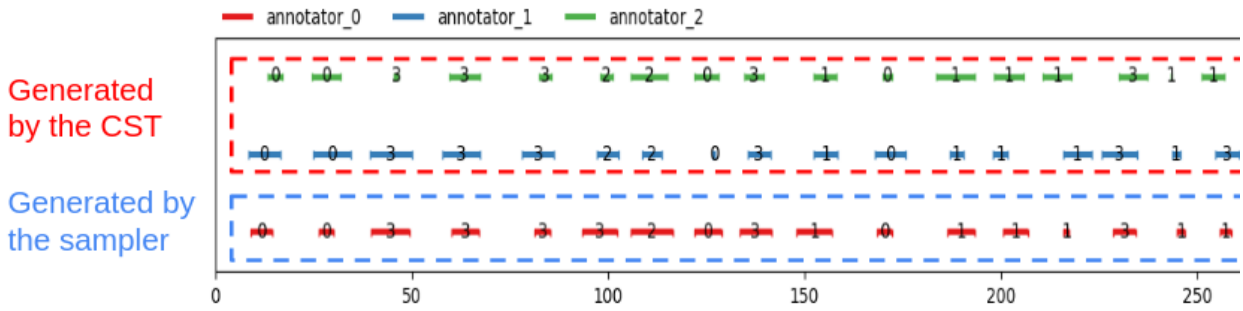
```
from pygamma_agreement import StatisticalContinuumSampler, CorpusShufflingTool, Continuum

sampler = StatisticalContinuumSampler()
# avg stands for average, std stands for standart deviation. All values are generated␣
↪using normal distribution.
sampler.init_sampling_custom(["annotator_ref"],
                             avg_num_units_per_annotator=50, std_num_units_per_
↪annotator=10,
                             avg_duration=15, std_duration=3,
                             avg_gap=5, std_gap=1,
                             categories=["Speaker1", "Speaker2", "Speaker3"],
                             categories_weight=[0.5, 0.3, 0.2])  # Proportions of␣
↪annotations per speaker
reference_continuum: Continuum = sampler.sample_from_continuum
```

You could also measure the behavior of the gamma with your dissimilarity by tweaking the values in the sampler. Now, let's use the corpus shuffling tool to generate a continuum with several annotators, with the selected errors with a given magnitude m:

```
cst = CorpusShufflingTool(magnitude=m,  # m is a float in [0, 1]
                          reference_continuum=reference_continuum)
generated_continuum: Continuum = cst.corpus_shuffle(
                                     ["Annotator1", "Annotator2", "Annotator3"],
                                     shift=True,  # annotations are randomly translated␣
↪proportionally to m
                                     false_pos=True,  # random annotations are added,␣
↪amount propotional to m
                                     false_neg=True,  # random annotations are discarded,
↪ amount propotional to m
                                     split=True,  # segments are splitted in two, number␣
↪of splits propotional to m
                                     cat_shuffle=True,  # annotation categories are␣
↪changed, amount propotional to m
                                     include_ref=False) # If true, copies the reference's␣
↪annotations.
```

Now you have a beautiful continuum, ready to be worked on !




```
dissim: AbstractDissimilarity
gamma_results = generated_continuum.compute_gamma(dissim)
```

Beware that a lot of randomness is involved in gamma computation and continuum generation, so you might want to seed using `np.seed` if you're making graphs. Averaging several values computed from continua generated with the same parameters might be better too.

1.5 Issues

1.5.1 Gamma Software issues

We observed early on in pygamma-agreement development that we weren't able to perfectly match [mathet2015]'s results from their closed-source Java implementation (the "Gamma Software"). In an effort to understand these discrepancies between our implementation of the gamma measure and theirs, we decompiled their application and carefully studied its code. This allowed us to find a number of small (yet significant) implementation details that were either undocumented or arbitrary.

In this section, we list off all the details that might make our calculation of the gamma-agreement deviate from the Java implementation's calculation, and explain what our own implementation choice is.

Warning: What we call "undocumented" are choices of implementation found in the Gamma Software that are not mentionned or explained in [mathet2015] or [mathet2018]. We made the choice of replicating some of those in pygamma-agreement, and not others,

1. Average number of annotations per annotator

In [mathet2015], section 4.3, a value is defined as such:

> "let $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ be the average number of annotations per annotator"

This value is involved in the computation of the disorder of an alignment.

In the Gamma Software: an int-instead-of-float division transforms this value into $\bar{x} = \lfloor \frac{\sum_{i=1}^{n} x_i}{n} \rfloor$.

In pygamma-agreement: We chose not to replicate this small discrepancy as it seemed like a bug, and didn't weight too much on the value of the gamma agreement.

Although it has no influence over which alignment will be considered the best alignment, it slighly changes the value of the disorders, which tweaks the gamma agreement for small continua.

2. Minimal distance between pivots

In [mathet2015], section 5.2.1, *Mathet et Al.* explain their method of sampling continuua by shuffling the reference continuum using random shift positions; and they specify a constraint on those positions :

> "To limit this phenomenon, we do not allow the distance between two shifts to be less than the average length of units."

In the Gamma Software: The value used for this minimal distance is actually **half** the average length of units.

In pygamma-agreement: We decided to include this discrepancy in the *ShuffleContinuumSampler* as it is designed to mimic the java implementation's, as opposed to our *StatisticalContinuumSampler* used by default by `pygamma-agreement`.

3. Pairing confidence

In [mathet2018], section 4.2.3, the pairing confidence of a pair of annotations is defined as such:

> "for $pair_i = (u_j, u_k)$, $p_i = max(0, 1 - d_{pos}(u_j, u_k))$"

In the Gamma Software: Their implementation of this formula uses a combined dissimilarity $d_{\alpha,\beta} = \alpha d_{pos} + \beta d_{cat}$, which transforms the formula for the pairing confidence this way: "$pair_i = (u_j, u_k)$, $p_i = max(0, 1 - \alpha \times d_{pos}(u_j, u_k))$".

In pygamma-agreement: Although it looked a lot like a bug, ignoring it makes the values of gamma-cat/k too different from those of the gamma software. We chose to include the alpha factor, as setting it to *1.0* can remove the discrepancy :

```
dissimilarity = CombinedCategoricalDissimilarity(alpha=3.0, # Set any alpha value you␣
↪want
                                                  beta=2.0,
                                                  delta_empty=1.0)

gamma_results = continuum.compute_gamma(dissimilarity)
dissimilarity.alpha = 1.0  # gamma_results stores the dissimilarity used for computing␣
↪the
                           # best alignments, as it is needed for computing gamma-cat
print(f"gamma-cat is {gamma_results.gamma_cat}")  # Gamma-k can also be influenced by␣
↪alpha
dissimilarity.alpha = 3.0  # Add this line if you want to reuse the dissimilarity with␣
↪alpha = 3
```

4. Best alignment

The Mixed Integer Programming solvers used in *pygamma-agreement* not being the same as the one used by the Gamma-Software, it is possible that the best alignments found by both software are different if multiple best alignments with the same disorder exist.

In the Gamma Software: The MIP solver used is `liblpsolve`

In pygamma-agreement: The MIP solver used is `GLPK`, or the faster `CBC` if it is installed.

Although this doesn't weight on the value of gamma, it slightly does on gamma-cat and gamma-k's. Thus, there is no way to obtain for sure the same results as the Gamma Software for gamma-cat/k.

1.5.2 How to obtain the results from the Gamma Software

This part explains how one can obtain an *almost* similar output as the Gamma Software using `pygamma-agreement`. The two main differences being :

Sampler

The sampler `pygamma-agreement` uses by default is **not** the one described in [mathet2015]. Our sampler collects statistical data about the input continuum (averages / standard deviation of several values such as length of annotations), used then to generate the samples. We made this choice because we felt that their sampler, which simply re-shuffles the input continuum, was unconvincing for the need of 'true' randomness.

To re-activate their sampler, you can use the `--mathet-sampler` (or `-m`) option when using the command line, or manually set the sampler used for computing the gamma agreement in python :

```
from pygamma_agreement import ShuffleContinuumSampler
...
gamma_results = continuum.compute_gamma(sampler=ShuffleContinuumSampler(),
                                        precision_level=0.01)
```

Alpha value

The Gamma Software uses $\alpha = 3$ in the combined categorical dissimilarity.

To set it in the command line interface, simply use the `--alpha 3` (or `-a 3`) option. In python, you need to manually create the combined categorical dissimilarity with the `alpha=3` parameter.

```
dissim = CombinedCategoricalDissimilarity(alpha=3)
gamma_results = continuum.compute_gamma(dissim,
                                        sampler=ShuffleContinuumSampler(),
                                        precision_level=0.01)
```

1.5.3 Bugs in former versions of pygamma-agreement

This section adresses fatal errors in release *0.1.6* of `pygamma-agreement`, whose consequences were a wrong output for gamma or other values. Those have been fixed in version *1.0.0*.

1. Average number of annotations per annotator

In [mathet2015], section 4.3, a value is defined as such:

> "let $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ be the average number of annotations per annotator"

A misreading made us interpret this value as the **total number of annotations** in the continuum. Thus, the values calculated by `pygamma-agreement` were strongly impacted (a difference as big as *0.2* for small continua).

2. Minimal distance between pivots

In [mathet2015], section 5.2.1, *Mathet et Al.* explain their method of sampling continuua by shuffling the reference continuum using random shift positions; and they specify a constraint on those positions :

> "To limit this phenomenon, we do not allow the distance between two shifts to be less than the average length of units."

In the previous version of the library, we overlooked this specificity of the sampling algorithm, which made the gamma values slightly bigger than expected (even after correction of the previous, far more impactful error).

1.6 Performance

This section aims to describe and explain the performances of the `pygamma-agreement` library in terms of time and memory usage.

1.6.1 Computational complexity

For a continuum with :

- p annotators
- n annotations per annotator
- N random samples involved (depends on the required precision)

The computational complexity of `Continuum.compute_gamma()` is :

$$C(N, n, p) = O(N \times n^p)$$

We're aware that for a high amount of annotators, the computation takes a lot of time and cannot be viable for realistic input.

The theorical complexity cannot be reduced, however we have found a *workaround* that sacrifices precision for a **significant** gain in complexity.

Moreover, the computation of the sample's dissimilarity is parellelized, which means that the complexity can be reduced to at best $O(s \times N \times n^p/c)$ with c CPUs.

1.6.2 Fast option

The `Continuum.compute_gamma()` method allows to set the *"fast"* option, which uses a different algorithm for determining the best alignment of a disorder. Although there is no found theory to back the precision of the algorithm, we have found out that it gives the **exact** results for the best alignments for real data, and a good approximation with continua generated specifically to mess with the algorithm.

Let's explain how this works with the notations of [mathet2015]. The algorithm creates an alignment recursively, each time eliminating a certain number of units in the continuum : for $\mathcal{U}$ a set of units, $\mathcal{A}$ the set of annotators, d a dissimilarity, $\mathcal{UA}$ the set of unitary alignments of $\mathcal{U}$ and $\mathcal{UA}_u$ the set of unitary alignments of $\mathcal{U}$ containing $u \in \mathcal{U}$:

- Let $\breve{a}(\mathcal{U})$ be any one of its best alignments.
- For x a real number, let $\mathcal{U}[x] = \{u \in \mathcal{U} \mid end(u) \leqslant x\}$.
- For w an integer, let $\mathcal{U}_w = \mathcal{U}[min\{x \mid |\mathcal{U}[x]| \geqslant w \times |\mathcal{A}|\}]$, the "Head of size w of $\mathcal{U}$".

- For w an integer, let $\epsilon_w(\mathcal{U}) = \{u \in \mathcal{U} \mid \exists u' \in \mathcal{U}_w, d(u, u') \leq \Delta_\emptyset|\mathcal{A}|\}$ and $\mathcal{U}_w^+ = \mathcal{U}_w \cup \epsilon_w(\mathcal{U})$, the "extended head of size w of $\mathcal{U}$".

Then, let us define the windowed best alignment of size w of $\mathcal{U}$ ($\mathcal{UA}$ being the set of unitary alignments of $\mathcal{U}$) :

$$\breve{a}_w(\mathcal{U}) = \breve{a}(\mathcal{U}_w^+) \cap \mathcal{U}_w\mathcal{A}$$

Finally, we can define the "fast alignment of window w" $\breve{\alpha}_w(\mathcal{U})$ recursively :

$$\breve{f}_w(\emptyset) = \emptyset$$

$$\breve{f}_w(\mathcal{U}) = \breve{a}_w(\mathcal{U}) \cup \breve{f}_w(\mathcal{U} \setminus \{u \in \mathcal{U} \mid \exists \bar{a} \in \breve{a}_w(\mathcal{U}), \bar{a} \in \mathcal{UA}_u\})$$

To clarify, here is an animation of how the "fast-alignment" is found :

The fast-gamma is simply the gamma with the best alignment of a each continuum (input and samples) replaced by the fast alignment with a certain window size.

To put words onto the fast-alignment algorithm, it consists of computing the windowed best alignment of size w of the continuum (using the classical MIP method), saving its unitary alignments & discarding the units saved from the continuum, and doing it again until the continuum is empty.

It uses the fact that alignments are made using locality of annotations, so it is only precise with a dissimilarity that mainly takes positionning into account. Results are still good with a combined categorical dissimilarity where $\alpha = 2 \times \beta$, and the window size doesn't seem to affect precision.

The fast gamma algorithm uses some sort of *"windowing"* of the continuum, and we have determined an approximation of its computational complexity, with w the number of annotation per annotator in a window :

$$C(w, N, n, p) = N \times (\frac{n}{w} \times (\lambda \times (w + s)^p + D(n, p, w, s))$$

With :

- D an additional complexity per window that we will not detail here.
- λ the *"numba factor"*, i.e the gain of speed obtained by using compiled numba functions, that we have estimated.
- s the *"additionnal window size"*, which is higher when the continuum has lots of overlapping.

This becomes a lot more interesting when the amount of annotations grows. One important thing to notice is that this complexity can be minimized with the right window size.

Here are the performance comparisons between the two algorithms :

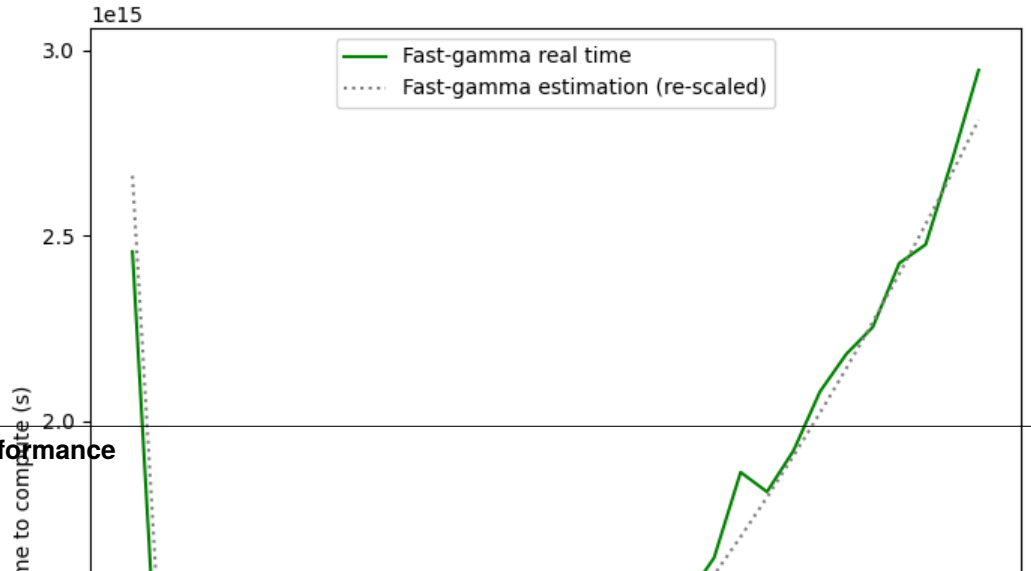


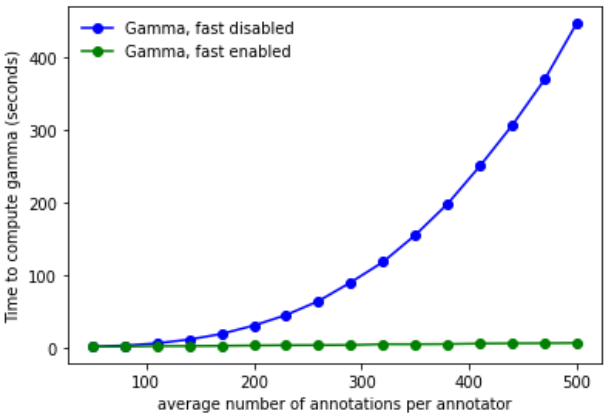
Fig. 3: **2 annotators**

As demonstrated, the gain in computation time provided by the *fast-gamma* optimization is very interesting, and the precision is more than sufficient, so we strongly advise to use the *fast-gamma* at all times.

So far, we haven't found a proof that the value given by the fast-gamma algorithm is precise.

However, we also haven't managed to find a case where the inaccuracy is significant.

Thus, for real (i.e. natural) input, it is established from experience that fast-gamma is more than reliable : it is advised to prioritize it since the gain in computing time is significant.




```
Average error of fast_gamma : 0.00025041645022767095
```

Fig. 4: **3 annotators**

1.7 The Soft-Gamma-Agreement : an alternate measure to gamma

To complete the gamma-agreement, we have added an option called "soft" gamma. It is a small tweak to the measure created with the goal of doing exactly what gamma does, except it reduces the disagreement caused by splits in annotations.

The idea behind this concept is to make use of the gamma agreement with machine learning models, since most of the existing ones are prone to produce of lot more splitted annotations than human annotators.

1.7.1 How to use soft-gamma

The soft-gamma measure, for the user at least, works exactly like gamma :

```
continuum = pa.Continuum.from_csv("tests/data/AlexPaulSuzan.csv")
dissim = pa.CombinedCategoricalDissimilarity(delta_empty=1,
                                             alpha=0.75,
                                             beta=0.25)
gamma_results = continuum.compute_gamma(dissim, soft=True)

print(f"gamma = {gamma_results.gamma}")
pa.show_alignment(gamma_results.best_alignment)
```

The only difference will be the look of the resulting best alignment, as well as the gamma value. This new gamma can be higher than the normal gamma (it is very unlikely to be lower). The more splitted annotations the input continuum contains, the wider the differences between the two measures will be.

Here is a comparison of those two measures, with two types of errors generated by the corpus shuffling tool :

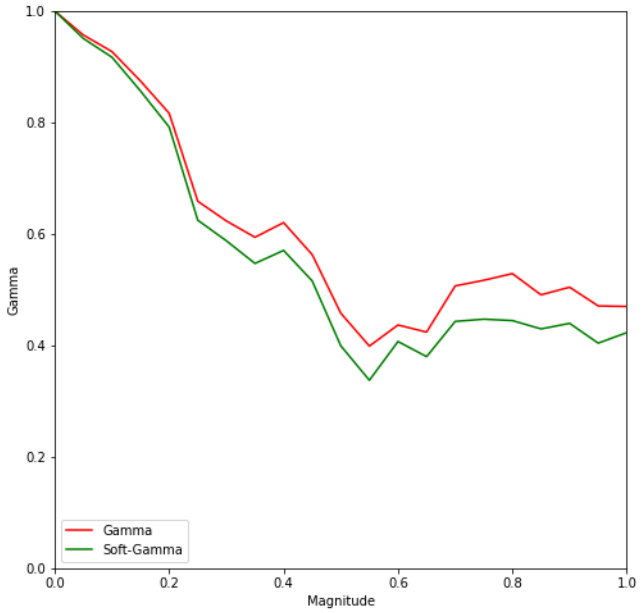


Fig. 5: **Shifted annotations, 3 annotators.**

For **shifts**, a magnitude of p means every start and end of unit is shifted by a random value between 0 and $p \times l$, where l is the duration of the unit.

For **splits**, a magnitude of p means that for an annotator who has annotated n segments, $p \times n \times 2$ units are successively chosen at random to be split at a random time. A unit can be splitted more than once.

As pointed before, the gamma-agreement is very sensitive to split annotations. Soft-gamma was created with the intent of reducing the penalty of splits while keeping a similar value if none are involved.

1.7.2 What is soft-gamma ?

This section will explain the differences between the two measures. Beware that it requires a bit of knowledge about the gamma-agreement. You can learn more about it in the "Principles" section :

> "**A unitary alignment** is a tuple of *units*, each belonging to a unique annotator. For a given *continuum* containing annotations from n different annotators, the tuple will *have* to be of length n. It represents a

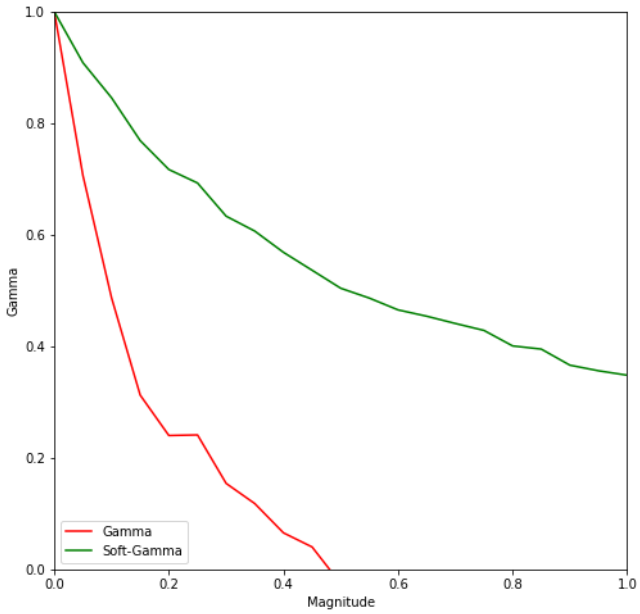


Fig. 6: **Splitted annotations, 3 annotators.**

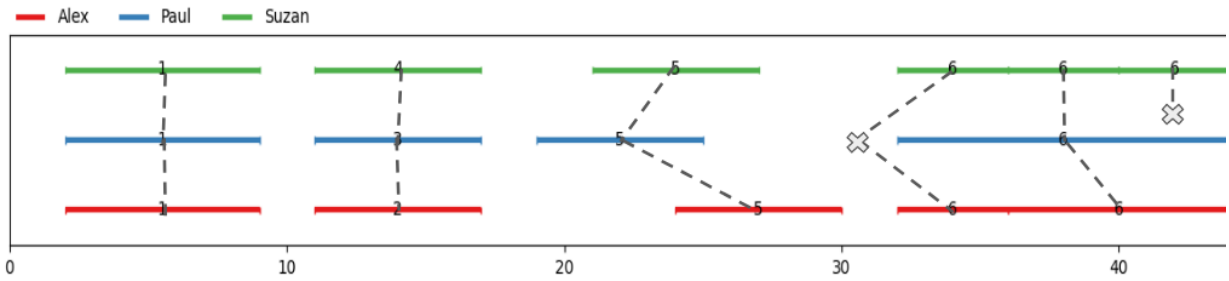
"match" (or the hypothesis of an agreement) between units of different annotators. A unitary alignment can contain *empty units*, which are "fake" (and null) annotations inserted to represent the absence of corresponding annotation for one or more annotators."

The main difference between soft-gamma and gamma is their definition of an **Alignment**, which the algorithms find the best possible based on the dissimilarity measure.

for Gamma :

"**An alignment** is a set of *unitary alignments* that constitute a *partition* of a continuum. This means that each and every unit of each annotator from the partitioned continuum can be found in the alignment **once, and only once**. "

To illustrate, let's visualize an alignment :

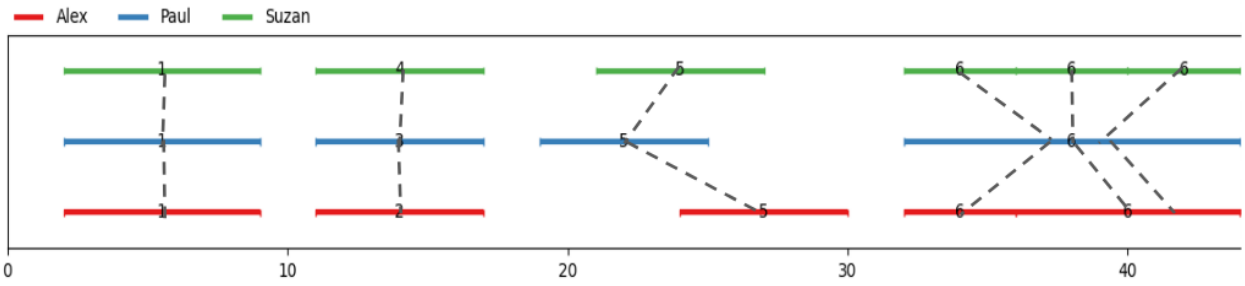


Every unit must be present **exactly** once, meaning splits create alignments with empty units, resulting in an additionnal disorder cost of more than $\Delta_\emptyset \times n$ per split.

for Soft-Gamma :

"**A soft-alignment** is a set of *unitary alignments* that constitute a *superset* of a continuum. This means that each and every unit of each annotator from the partitioned continuum can be found in the alignment **at least once**."

To illustrate, let's visualize an alignment :



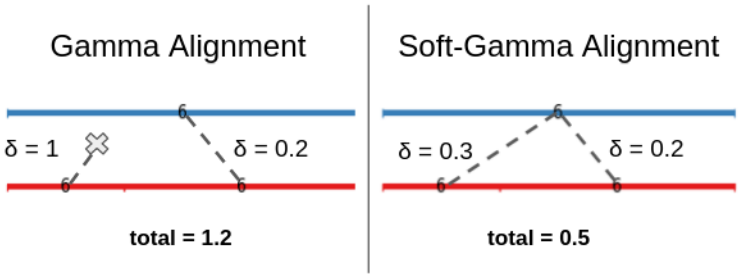
Every unit must be present **at least** once, meaning splits create multi-aligned units, with significantly lower cost. Thus, this alignment will be preferred as the one before.

This definition ensures empty units are added to mark up a false negative and not to complete the partition of the continuum with unitary alignments.

1.7.3 How does soft-gamma reduce split cost ?

The important effect of a soft-alignment is that it limits the occurence of "empty units" in the best possible alignment. With the 'gamma' definition of an alignment, in most cases, splits cause empty units to appear in the best alignments.

This is because with the constraint of having every unit (except empty ones) appear exactly once in the best alignment, the algorithm will naturally 'fill the gaps' with empty units when there's a split : if an annotator has written down two annotations where the other has annotated only once, an empty unit **will** make its way into the best alignment.



With the soft-gamma definition, the resulting cost of such a split will only be the sum of the two possible alignment, which is usally a lot lower than the cost of using the empty unit. Naturally, this depends on the nature of the dissimilarity used : one must ensure that it isn't more advantageous to use lots of empty units.

1.7.4 Combining soft-gamma with gamma-cat

Combining a "soft"-alignment with the gamma-cat or gamma-k measure (which uses the best alignment determined by the algorithm) is very much possible. Simply do just as you would with the normal gamma :

```
continuum = pa.Continuum.from_csv("tests/data/AlexPaulSuzan.csv")
dissim = pa.CombinedCategoricalDissimilarity(delta_empty=1,
                                             alpha=0.75,
                                             beta=0.25)
gamma_results = continuum.compute_gamma(dissim, soft=True)

print(f"gamma-cat = {gamma_results.gamma_cat}")
print(f"gamma-k('7') = {gamma_results.gamma_k('7')}")
```

We would recomment this version of gamma-cat instead of the classical one, since the less overwhelming amount of null units seems to suit more the objective of gamma-cat, which is only determining the exactitude of **categorization** of a continuum, and splitting causes the appearance of null units which is equivalent to not having an unexisting category and thus categorizing wrong even when two annotations with the same category are indeed overlapping.

1.8 API Reference

1.8.1 Data structures

class pygamma_agreement.**Unit**(*segment: Segment*, *annotation: Optional[str] = None*)

Represents an annotated unit, e.g., a time segment and (optionally) a text annotation. Can be sorted or used in a set. If two units share the same time segment, they're sorted alphabetically using their annotation. The *None* annotation is first in the "alphabet"

```
>>> new_unit = Unit(segment=Segment(17.5, 21.3), annotation='Verb')
>>> new_unit.segment.start, new_unit.segment.end
17.5, 21.3
>>> new_unit.annotation
'Verb'
```

__delattr__(*name*)

Implement delattr(self, name).

__eq__(*other*)

Return self==value.

__ge__(*other*, *NotImplemented=NotImplemented*)

Return a >= b. Computed by @total_ordering from (not a < b).

__gt__(*other*, *NotImplemented=NotImplemented*)

Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__hash__()

Return hash(self).

__init__(*segment: Segment*, *annotation: Optional[str] = None*) → None

__le__(*other*, *NotImplemented=NotImplemented*)

Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

__lt__(*other:* Unit)

Return self<value.

__repr__()

Return repr(self).

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

class pygamma_agreement.**Continuum**(*uri: Optional[str] = None*)

Representation of a continuum, i.e a set of annotated segments by multiple annotators. It is implemented as a dictionnary of sets (all sorted) :

```
{'annotator1':  {unit1, ...}, ...}
```

__add__(*other:* Continuum)

Same as a "not-in-place" merge.

Parameters

other (Continuum) – the continuum to merge into *self*

__bool__()

Truthiness, basically tests for emptiness

```
>>> if continuum:
...     # continuum is not empty
... else:
...     # continuum is empty
```

__eq__(*other:* Continuum)

Two continua are equal if and only if all their annotators and all their units are strictly equal

__getitem__(*keys: Union[str, Tuple[str, int]]*) → Union[SortedSet, *Unit*]

Get the set of annotations from an annotator, or a specific annotation. (Deep copies are returned to ensure some constraints cannot be violated)

```
>>> continuum['Alex']
SortedSet([Unit(segment=<Segment(2, 9)>, annotation='1'), Unit(segment=
↪<Segment(11, 17)>, ...
>>> continuum['Alex', 0]
Unit(segment=<Segment(2, 9)>, annotation='1')
```

Parameters
keys (*Annotator or Annotator,int*) –

Raises
KeyError –

__hash__ = None

__init__(*uri: Optional[str] = None*)

Default constructor.

Parameters
uri (*optional str*) – name of annotated resource (e.g. audio or video file)

__iter__() → Generator[Tuple[str, *Unit*], None, None]

Iterates over (annotator, unit) tuples for every unit in the continuum.

__ne__(*other:* Continuum)

Return self!=value.

add(*annotator: str*, *segment: Segment*, *annotation: Optional[str] = None*)

Add a segment to the continuum

Parameters

- **annotator** (*Annotator (str)*) – The annotator that produced the added annotation
- **segment** (*pyannote.core.Segment*) – The segment for that annotation
- **annotation** (*optional str*) – That segment's annotation, if any.

add_annotation(*annotator: str*, *annotation: Annotation*)

Add a full pyannote annotation to the continuum.

Parameters

- **annotator** (*Annotator (str)*) – A string id for the annotator who produced that annotation.
- **annotation** (*pyannote.core.Annotation*) – A pyannote *Annotation* object. If a label is present for a given segment, it will be considered as that label's annotation.

add_annotator(*annotator: str*)

Adds the annotator to the set, with no annotated segment. Does nothing if already present.

add_elan(*annotator: str*, *eaf_path: Union[str, Path]*, *selected_tiers: Optional[List[str]] = None*, *use_tier_as_annotation: bool = False*)

Add an Elan (.eaf) file's content to the Continuum

Parameters

- **annotator** (*Annotator (str)*) – A string id for the annotator who produced that ELAN file.
- **eaf_path** (*Path* or str) – Path to the .eaf (ELAN) file.
- **selected_tiers** (*optional list of str*) – If set, will drop tiers that are not contained in this list.
- **use_tier_as_annotation** (*optional bool*) – If True, the annotation for each non-empty interval will be the name of its parent Tier.

add_textgrid(*annotator: str*, *tg_path: Union[str, Path]*, *selected_tiers: Optional[List[str]] = None*, *use_tier_as_annotation: bool = False*)

Add a textgrid file's content to the Continuum

Parameters

- **annotator** (*Annotator (str)*) – A string id for the annotator who produced that TextGrid.
- **tg_path** (*Path* or str) – Path to the textgrid file.
- **selected_tiers** (*optional list of str*) – If set, will drop tiers that are not contained in this list.
- **use_tier_as_annotation** (*optional bool*) – If True, the annotation for each non-empty interval will be the name of its parent Tier.

add_timeline(*annotator: str*, *timeline: Timeline*)

Add a full pyannote timeline to the continuum.

Parameters

- **annotator** (*Annotator (str)*) – A string id for the annotator who produced that timeline.
- **timeline** (*pyannote.core.Timeline*) – A pyannote *Annotation* object. No annotation will be attached to segments.

property annotators: SortedSet

Returns a sorted set of the annotators in the Continuum

```
>>> self.annotators:
... SortedSet(["annotator_a", "annotator_b", "annot_ref"])
```

property avg_length_unit: float

Mean of the annotated segments' durations

property avg_num_annotations_per_annotator: float

Average number of annotated segments per annotator

property bounds: Tuple[float, float]

Bounds of the continuum. Initially defined as (0, 0), they grow as annotations are added.

property categories: SortedSet

Returns the (alphabetically) sorted set of all the continuum's annotations's categories.

property category_weights: SortedDict

Returns a dictionary where the keys are the categories in the continuum, and a key's value is the proportion of occurrence of the category in the continuum.

compute_gamma(*dissimilarity: Optional[AbstractDissimilarity] = None*, *n_samples: int = 30*, *precision_level: Optional[Union[float, typing_extensions.Literal[high, medium, low]]] = None*, *ground_truth_annotators: Optional[SortedSet] = None*, *sampler: AbstractContinuumSampler = None*, *fast: bool = False*, *soft: bool = False*) → *GammaResults*

Parameters

- **dissimilarity** (*AbstractDissimilarity, optional*) – dissimilarity instance. Used to compute the disorder between units. If not set, it defaults to the combined categorical dissimilarity with parameters taken from the java implementation.
- **n_samples** (*optional int*) – number of random continuum sampled from this continuum used to estimate the gamma measure
- **precision_level** (*optional float or "high", "medium", "low"*) – error percentage of the gamma estimation. If a literal precision level is passed (e.g. "medium"), the corresponding numerical value will be used (high: 1%, medium: 2%, low : 5%)
- **ground_truth_annotators** (*SortedSet of str*) – if set, the random continuua will only be sampled from these annotators. This should be used when you want to compare a prediction against some ground truth annotation.
- **sampler** (*AbstractContinuumSampler*) – Sampler object, which implements a sampling strategy for creating random continuua used to calculate the expected disorder. If not set, defaults to the Statistical continuum sampler
- **fast** – Sets the algorithm to the much faster fast-gamma. It's supposed to be less precise than the "canonical" algorithm from Mathet 2015, but usually isn't. Performance gains and precision are explained in the Performance section of the documentation.
- **soft** – Activate soft-gamma, an alternative measure that uses a slighlty different definition of an alignment. For further information, please consult the 'Soft-Gamma' section of the documentation. Incompatible with fast-gamma : raises an error if both 'fast' and 'soft' are set to True.

copy() → *Continuum*

Makes a copy of the current continuum.

Returns
continuum

Return type
Continuum

copy_flush() → *Continuum*

Returns a copy of the continuum without any annotators/annotations, but with every other information

classmethod from_csv(*path: Union[str, Path]*, *discard_invalid_rows=True*, *delimiter: str = ','*)

Load annotations from a CSV file , with structure annotator, category, segment_start, segment_end.

> **Warning:** The CSV file mustn't have any header

Parameters

- **path** (*Path or str*) – Path to the CSV file storing annotations
- **discard_invalid_rows** (*bool*) – If set, every invalid row is ignored when parsing the file.

- **delimiter** (*str*) – CSV columns delimiter. Defaults to ‘,’

Returns
New continuum object loaded from the CSV

Return type
Continuum

classmethod from_rttm(*path: Union[str, Path]*) → *Continuum*

Load annotations from a RTTM file. The file name field will be used as an annotation’s annotator

Parameters
path (*Path or str*) – Path to the RTTM file storing annotations

Returns
continuum – New continuum object loaded from the RTTM file

Return type
Continuum

get_best_alignment(*dissimilarity:* AbstractDissimilarity) → *Alignment*

Returns the best alignment of the continuum for the given dissimilarity. This alignment comes with the associated disorder, so you can obtain it in constant time with alignment.disorder. Beware that the computational complexity of the algorithm is very high ($O(p_1 \times p_2 \times ... \times p_n)$ where p_i is the number of annotations of annotator i).

Parameters
dissimilarity (AbstractDissimilarity) – the dissimilarity that will be used to compute unit-to-unit disorder.

get_fast_alignment(*dissimilarity:* AbstractDissimilarity, *window_size: int*) → *Alignment*

Returns an ‘approximation’ of the best alignment (Very likely to be the actual best alignment for continua with limited overlapping)

get_first_window(*dissimilarity:* AbstractDissimilarity, *w: int = 1*) → Tuple[*Continuum*, float]

Returns a tuple (continuum, x_limit), where :

- Before x_limit, there are the (w * nb_annotators) leftmost annotations of the continuum.
- After x_limit, there are (approximately) all the annotations from the continuum that have a dissimilarity lower than (delta_empty * nb_annotators) with the annotations before x_limit.

iter_annotator(*annotator: str*) → Generator[*Unit*, None, None]

Iterates over the annotations of the given annotator.

Raises
KeyError – If the annotators is not on this continuum.

iterunits(*annotator: str*)

Iterate over units from the given annotator (in chronological and alphabetical order if annotations are present)

```
>>> for unit in self.iterunits("Max"):
...     # do something with the unit
```

property max_num_annotations_per_annotator

The maximum number of annotated segments an annotator has in this continuum

measure_best_window_size(*dissimilarity:* AbstractDissimilarity)

Sets the best window size for computing the fast-gamma of this continuum, by using the sampling the computing complexity function.

merge(*continuum:* Continuum, *in_place: bool = False*) → Optional[*Continuum*]

Merge two Continuua together. Units from the same annotators are also merged together (with the usual order of units).

Parameters

- **continuum** (Continuum) – other continuum to merge into the current one.
- **in_place** (*bool*) – If set to true, the merge is done in place, and the current continuum (self) is the one being modified. A new continuum resulting in the merge is returned otherwise.

Returns
Continuum, optional

Return type
Returns the merged copy if in_place is set to True.

property num_annotators: int

Number of annotators

property num_units: int

Total number of units in the continuum.

remove(*annotator: str*, *unit:* Unit)

Removes the given unit from the given annotator's annotations. Keeps the bounds of the continuum as they are. :raises KeyError: if the unit is not from the annotator's annotations.

reset_bounds()

Resets the bounds of the continuum (used in displaying and/or sampling) to the start of leftmost annotation and the end of rightmost annotation.

class pygamma_agreement.**UnitaryAlignment**(*n_tuple: List[Tuple[str, Optional[*Unit*]]]*)

Unitary Alignment

Parameters
n_tuple – n-tuple where n is the number of annotators of the continuum This is a list of (annotator, segment) couples

__init__(*n_tuple: List[Tuple[str, Optional[*Unit*]]]*)

property bounds

Start of leftmost unit and end of rightmost unit

compute_disorder(*dissimilarity:* AbstractDissimilarity)

Building a fake one-element alignment to compute the disorder

property disorder: float

Disorder of the alignment. Raises ValueError if self.compute_disorder(dissimilarity) hasn't been called before.

property nb_units

The number of non-empty units in the unitary alignment.

class pygamma_agreement.**Alignment**(*unitary_alignments: Iterable[*UnitaryAlignment*]*, *continuum: Optional[*Continuum*] = None*, *check_validity: bool = False*, *disorder: Optional[float] = None*)

__init__(*unitary_alignments: Iterable[*UnitaryAlignment*]*, *continuum: Optional[*Continuum*] = None*, *check_validity: bool = False*, *disorder: Optional[float] = None*)

Alignment constructor.

Parameters

- **unitary_alignments** – set of unitary alignments that make a partition of the set of units/segments
- **continuum** (*optional Continuum*) – Continuum where the alignment is from
- **check_validity** (*bool*) – Check the validity of that Alignment against the specified continuum
- **disorder** (*float, optional*) – If set, self.disorder returns it until a call to self.compute_disorder. It allows to make the most of the best alignment computation, that takes advantage of this value.

check(*continuum: Optional[*Continuum*] = None*)

Checks that an alignment is a valid partition of a Continuum. That is, that all annotations from the referenced continuum *can be found* in the alignment and can be found *only once*. Empty units are not taken into account.

Parameters

continuum (*optional Continuum*) – Continuum to check the alignment against. If none is specified, will try to use the one set at instanciation.

Raises

ValueError, SetPartitionError –

compute_disorder(*dissimilarity:* AbstractDissimilarity)

Recalculates the disorder of this alignment using the given dissimilarity computer. Usually not needed since most alignment are generated from a minimal disorder.

property disorder: float

returns: The disorder of the alignment. :rtype: float

gamma_k_disorder(*dissimilarity:* AbstractDissimilarity, *category: Optional[str]*) → float

Returns the gamma-k or gamma-cat metric disorder. (Exact implementation of the algorithm from section 4.2.5 of https://hal.archives-ouvertes.fr/hal-01712281)

Parameters

- **dissimilarity** (AbstractDissimilarity) – the dissimilarity measure to be used in the algorithm. Raises ValueError if it is not a combined categorical dissimilarity, as gamma-cat requires both positional and categorical dissimilarity.
- **category** – If set, the category to be used as reference for gamma-k. Leave it unset to compute the gamma-cat disorder.

class pygamma_agreement.**GammaResults**(*best_alignment:* Alignment, *chance_alignments: List[*Alignment*]*, *dissimilarity:* AbstractDissimilarity, *precision_level: Optional[float] = None*)

Gamma results object. Stores the information about a gamma measure computation, used for getting the values of measures from the gamma family (gamma, gamma-cat and gamma-k).

__eq__(*other*)

Return self==value.

__hash__ = None

__init__(*best_alignment: Alignment*, *chance_alignments: List[Alignment]*, *dissimilarity: AbstractDissimilarity*, *precision_level: Optional[float] = None*) → None

__repr__()

Return repr(self).

property alignments_nb

Number of unitary alignments in the best alignment.

property approx_gamma_range

Returns a tuple of the expected boundaries of the computed gamma, obtained using the expected disagreement and the precision level

property expected_disorder: float

Returns the expected disagreement for computed random samples, i.e., the mean of the sampled continuua's disorders

property gamma: float

Returns the gamma value

property gamma_cat: float

Returns the gamma-cat value

gamma_k(*category: str*) → float

Returns the gamma-k value for the given category

property n_samples

Number of samples used for computation of the expected disorder.

property observed_disorder: float

Returns the disorder of the computed best alignment, i.e, the observed disagreement.

1.8.2 Dissimilarities

class pygamma_agreement.**AbstractDissimilarity**(*categories: Optional[SortedSet] = None*, *delta_empty: float = 1.0*)

Function used to measure the difference between two annotations, using their positioning and categorization.

Parameters

- **delta_empty** (*float*) – Distance between a unit and a "null" unit. Defaults to 1.0
- **categories** (*SortedSet of str, optional*) – Labels of annotations involved. Some categories don't consider the actual content of the categories, so it is left optional.

__init__(*categories: Optional[SortedSet] = None*, *delta_empty: float = 1.0*)

abstract compile_d_mat() → Callable[[ndarray, ndarray], float]

Must set self.d_mat to the the cfunc (decorated with @dissimilarity_dec) function that corresponds to the unit-to-unit, (in arrays form) disorder given by the dissimilarity.

compute_disorder(*alignment:* Alignment) → ndarray

Returns the disorder of the given alignment.

abstract d(*unit1:* Unit, *unit2:* Unit)

Dissimilarity between two units as a real Unit object.

valid_alignments(*continuum:* Continuum) → Tuple[ndarray, ndarray]

Returns all the unitary alignment (in matricial form), and their disorders that could potentially be in the best alignment of the continuum (based on the criterium detailed in section 5.1.1 of the gamma paper (https://aclanthology.org/J15-3003.pdf).

class pygamma_agreement.**PositionalSporadicDissimilarity**(*delta_empty: float = 1.0*)

Positional-sporadic dissimilarity. Takes only the position of annotations into account. This distance is : * 0 when segments are equal * < delta_empty when segments completely overlap $A \cup B = A$ or B) * > delta_empty when segments are separated ($A \cap B = \emptyset$)

__init__(*delta_empty: float = 1.0*)

compile_d_mat()

Must set self.d_mat to the the cfunc (decorated with @dissimilarity_dec) function that corresponds to the unit-to-unit, (in arrays form) disorder given by the dissimilarity.

d(*unit1:* Unit, *unit2:* Unit)

Dissimilarity between two units as a real Unit object.

class pygamma_agreement.**CategoricalDissimilarity**(*categories: SortedSet*, *delta_empty: float = 1.0*)

Abstract base class for categorical dissimilarity.

__init__(*categories: SortedSet*, *delta_empty: float = 1.0*)

class pygamma_agreement.**AbsoluteCategoricalDissimilarity**(*delta_empty: float = 1.0*)

Basic categorical dissimilarity. Worth 0.0 when categories are identical, delta_empty otherwise.

__init__(*delta_empty: float = 1.0*)

compile_d_mat()

Must set self.d_mat to the the cfunc (decorated with @dissimilarity_dec) function that corresponds to the unit-to-unit, (in arrays form) disorder given by the dissimilarity.

d(*unit1:* Unit, *unit2:* Unit)

Dissimilarity between two units as a real Unit object.

class pygamma_agreement.**PrecomputedCategoricalDissimilarity**(*categories: SortedSet*, *matrix: ndarray*, *delta_empty: float = 1.0*)

Categorical dissimilarity with a provided matrix that contains all the category-to-category dissimilarity. The indexes of the matrix correspond to the **categories in alphabetical order**.

__init__(*categories: SortedSet*, *matrix: ndarray*, *delta_empty: float = 1.0*)

compile_d_mat()

Must set self.d_mat to the the cfunc (decorated with @dissimilarity_dec) function that corresponds to the unit-to-unit, (in arrays form) disorder given by the dissimilarity.

d(*unit1:* Unit, *unit2:* Unit)

Dissimilarity between two units as a real Unit object.

class pygamma_agreement.**OrdinalCategoricalDissimilarity**(*labels: Iterable[str]*, *p: Optional[Iterable[float]] = None*, *delta_empty=1.0*)

Categorical dissimilarity where each label is given a position on the real axis, and the disorder between categories of positions 'a' and 'b' being **|a - b|**/m * delta_empty with m the maximum position. If not provided, positions are 0, 1, 2...

__init__(*labels: Iterable[str]*, *p: Optional[Iterable[float]] = None*, *delta_empty=1.0*)

Parameters

- **labels** (*Iterable of str*) – The categories involved in the dissimilarity
- **p** (*Iterable of floats*) – The real numbers associated with each label, in the same order.

class pygamma_agreement.**NumericalCategoricalDissimilarity**(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

Categorical dissimilarity made for numerical categories (i.e a category is a float or int literal). The disorder between categories 'a' and 'b' being **|a - b|**/m * delta_empty with m the maximum category.

__init__(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

Parameters

- **labels** (*Iterable of str*) – The categories involved in the dissimilarity
- **p** (*Iterable of floats*) – The real numbers associated with each label, in the same order.

class pygamma_agreement.**LambdaCategoricalDissimilarity**(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

Categorical dissimilarity, whose values are precomputed from a (str, str) -> float function (the *cat_dissim_func* method) and the list of categories provided.

__init__(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

class pygamma_agreement.**LevenshteinCategoricalDissimilarity**(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

Precomputed categorical dissimilarity whose value is the proportional levenshtein distance between the category labels.

__init__(*labels: Iterable[str]*, *delta_empty: float = 1.0*)

class pygamma_agreement.**CombinedCategoricalDissimilarity**(*alpha: float = 1.0*, *beta: float = 1.0*, *delta_empty: float = 1.0*, *pos_dissim: Optional[AbstractDissimilarity] = None*, *cat_dissim: Optional[CategoricalDissimilarity] = None*)

This dissimilarity takes both positioning and categorizing of annotations into account. Combined categorical dissimilarity constructor. :param delta_empty: empty dissimilarity value. Defaults to 1. :type delta_empty: optional, float :param alpha: coefficient weighting the positional dissimilarity value.

Defaults to 1.

Parameters

- **beta** (*optional float*) – coefficient weighting the categorical dissimilarity value. Defaults to 1.

- **cat_dissim** (*optional, CategoricalDissimilarity*) – Categorical-only dissimilarity to be used. If not set, defaults to the absolute categorical dissimilarity.

__init__(*alpha: float = 1.0*, *beta: float = 1.0*, *delta_empty: float = 1.0*, *pos_dissim: Optional[AbstractDissimilarity] = None*, *cat_dissim: Optional[CategoricalDissimilarity] = None*)

compile_d_mat()

Must set self.d_mat to the the cfunc (decorated with @dissimilarity_dec) function that corresponds to the unit-to-unit, (in arrays form) disorder given by the dissimilarity.

d(*unit1: Unit*, *unit2: Unit*)

Dissimilarity between two units as a real Unit object.

1.8.3 Samplers

class pygamma_agreement.**AbstractContinuumSampler**

Tool for generating sampled continuua from a reference continuum. Used to compute the "expected disorder" when calculating the gamma, using particular sampling techniques. Must be initalized (with self.init_sampling for instance)

__init__()

Super constructor, sets everything to None since a call to init_sampling to set parameters is mandatory.

init_sampling(*reference_continuum: Continuum*, *ground_truth_annotators: Optional[Iterable[str]] = None*)

Parameters

- **reference_continuum** (*Continuum*) – the continuum that will be shuffled into the samples
- **ground_truth_annotators** (*iterable of str, optional*) – the set of annotators (from the reference) that will be considered for sampling

abstract property sample_from_continuum: *Continuum*

Returns a shuffled continuum based on the reference. Everything in the generated sample is at least a copy.

Raises

ValueError: – if *init_sampling* or another initalization method hasn't been called before.

class pygamma_agreement.**ShuffleContinuumSampler**(*pivot_type: typing_extensions.Literal[float_pivot, int_pivot] = 'int_pivot'*)

This continuum sampler uses the methods used in gamma-software, ie those described in gamma-paper : https://www.aclweb.org/anthology/J15-3003.pdf, section 5.2. and implemented in the GammaSoftware.

__init__(*pivot_type: typing_extensions.Literal[float_pivot, int_pivot] = 'int_pivot'*)

This constructor allows to set the pivot type to int or float. Defaults to int to match the java implementation.

init_sampling(*reference_continuum: Continuum*, *ground_truth_annotators: Optional[Iterable[str]] = None*)

Parameters

- **reference_continuum** (*Continuum*) – the continuum that will be shuffled into the samples
- **ground_truth_annotators** (*iterable of str, optional*) – the set of annotators (from the reference) that will be considered for sampling

property **sample_from_continuum**: *Continuum*

Returns a shuffled continuum based on the reference. Everything in the generated sample is at least a copy.

Raises

ValueError: – if *init_sampling* or another initalization method hasn't been called before.

class pygamma_agreement.**StatisticalContinuumSampler**

This sampler creates continua using the average and standard deviation of :

- The number of annotations per annotator
- The gap between two of an annotator's annotations
- The duration of the annotations' segments

The sample is thus created by computing normal distributions using these parameters.

It also requires the probability of occurence of each annotations category. You can either initalize sampling with custom values or with a reference continuum.

init_sampling(*reference_continuum:* Continuum, *ground_truth_annotators: Optional[Iterable[str]] = None*)

Sets the sampling parameters using statistical values obtained from the reference continuum.

Parameters

- **reference_continuum** (Continuum) – the continuum that will be shuffled into the samples
- **ground_truth_annotators** (*iterable of str, optional*) – the set of annotators (from the reference) that will be considered for sampling

init_sampling_custom(*annotators: Iterable[str], avg_num_units_per_annotator: float, std_num_units_per_annotator: float, avg_gap: float, std_gap: float, avg_duration: float, std_duration: float, categories: Iterable[str], categories_weight: Optional[Iterable[float]] = None*)

Parameters

- **annotators** – the annotators that will be involved in the samples
- **avg_num_units_per_annotator** (*float, optional*) – average number of units per annotator
- **std_num_units_per_annotator** (*float, optional*) – standard deviation of the number of units per annotator
- **avg_gap** (*float, optional*) – average gap between two of an annotator's annotations
- **std_gap** (*float, optional*) – standard deviation of the gap between two of an annotator's annotations
- **avg_duration** (*float, optional*) – average duration of an annotation
- **std_duration** (*float, optional*) – standard deviation of the duration of an annotation
- **categories** (*np.array[str, 1d]*) – The possible categories of the annotations
- **categories_weight** (*np.array[float, 1d], optional*) – The probability of occurence of each category. Can raise errors if len(categories) != len(category_weights) and category_weights.sum() != 1.0. If not set, every category is equiprobable.

property sample_from_continuum: *Continuum*

Returns a shuffled continuum based on the reference. Everything in the generated sample is at least a copy.

Raises

ValueError: – if *init_sampling* or another initalization method hasn't been called before.

1.8.4 Corpus Shuffling Tool

class pygamma_agreement.**CorpusShufflingTool**(*magnitude: float*, *reference_continuum:* Continuum, *categories: Optional[Iterable[str]] = None*)

Corpus shuffling tool as detailed in section 6.3 of the gamma paper (https://www.aclweb.org/anthology/J15-3003.pdf#page=30).

__init__(*magnitude: float*, *reference_continuum:* Continuum, *categories: Optional[Iterable[str]] = None*)

Parameters

- **magnitude** – magnitude m of the cst (cf gamma paper)
- **reference_continuum** – this continuum will serve as reference for the tweaks made by the corpus shuffling tool.
- **categories** – this is used to consider additionnal categories when shuffling the corpus, in the eventuality that the reference continuum does not contain any unit of a possible category.

category_shuffle(*continuum:* Continuum, *overlapping_fun: Optional[Callable[[str, str], float]] = None*, *prevalence: bool = False*)

Shuffles the categories of the annotations in the given continuum using the process described in section 3.3.5 of https://hal.archives-ouvertes.fr/hal-00769639/. :param overlapping_fun: gives the "categorical distance" between two annotations, which is taken into account when provided.

(the lower the distance between categories, the higher the chance one will be changed into the other).

Parameters

prevalence – specify whether or not to consider the proportion of presence of each category in the reference.

corpus_shuffle(*annotators: Union[int, Iterable[str]]*, *shift: bool = False*, *false_pos: bool = False*, *false_neg: bool = False*, *split: bool = False*, *cat_shuffle: bool = False*, *include_ref: bool = False*) → *Continuum*

Generates a new shuffled corpus with the provided (or generated) reference annotation set, using the method described in 6.3 of the gamma paper, https://www.aclweb.org/anthology/J15-3003.pdf#page=30 (and missing elements described in another article : https://hal.archives-ouvertes.fr/hal-00769639/).

false_neg_shuffle(*continuum:* Continuum) → None

Tweaks the continuum by randomly removing units ("false negatives"). Every unit (for each annotator) have a probability equal to the magnitude of being removed. If this probability is one, a single random unit (for each annotator) will be left alone.

false_pos_shuffle(*continuum:* Continuum) → None

Tweaks the continuum by randomly adding "false positive" units. The number of added units per annotator is constant & proportionnal to the magnitude of the CST. The chosen category is random and depends on the probability of occurence of the category in the reference. The length of the segment is random (normal distribution) based on the average and standard deviation of those of the reference.

shift_shuffle(*continuum:* Continuum) → None

Tweaks the given continuum by shifting the ends of each segment, with uniformly distributed values of bounds proportionnal to the magnitude of the CST and the length of the segment.

splits_shuffle(*continuum:* Continuum)

Tweak the continuum by randomly splitting segments. Number of splits per annotator is constant & proportionnal to the magnitude of the CST and the number of units in the reference. A splitted segment can be re-splitted.

1.9 Changelog

1.9.1 Version 0.5.6 (2022-02-12) (@hadware)

- Added support for Python 3.10
- Bumped cvxopt required version to 1.2.7

1.9.2 Version 0.5.4 (2021-11-29)

- Fixed a bug in the *Continuum.merge* function when a continuun had annotators with no annotations (@valentinoli)
- Fixed a bug in the *Continuum.reset_bounds()* function when a continuun had annotators with no annotations (@valentinoli)
- Added *__eq__* and *__ne__* comparison magic methods to enable == and != operators on *Continuum* instances

1.9.3 Version 0.5.0 (2021-09-17) (@lfavre)

- **Added soft-gamma, an alternate inter-annotator agreement measure designed based on the gamma-agreement**
  - Extensive documentation about this measure and its uses.
- Minor bug fixes

1.9.4 Version 0.4.1 (2021-08-30) (@lfavre)

- Important bug fix : Some slicing error when the number of possible unitary alignments was to high.

1.9.5 Version 0.4.0 (2021-08-13) (@lfavre)

- **Fast-gamma option**
  - New algorithm that gives a satisfying approximation of the gamma-agreement, with a colossal gain in computing time and memory usage.
  - Detailed research, explanations and benchmarking about this algorithm are thoroughly detailed in the "Performances" section of the documentation
- Minor bug fixes

1.9.6 Version 0.3.0 (2021-08-06) (@lfavre)

- **New interface for dissimilarities.**
  - A real class structure, made with user-friendliness in mind
  - Making new or custom dissimilarities is now doable without copying huge chunks of numba code, supposedly without knowledge of the inner working of the library
  - The code for dissimilarities is overall clearer, more reliable and more maintainable
  - New natively available dissimilarities
- **Bug fixes**
  - Fixed many bugs that emerged from the sorted structures and a confusion when passing to a numba/numerical algorithmic environment
- **Optimizations :**
  - Memory usage of the gamma algorithm is now significantly lower than before
  - Unit-to-unit disorders are pre-computed during the best-alignment algorithm, which lowers the computation time
  - Multiprocessing has been replaced by multithreading, for additionnal memory usage, computation time and simplicity.
  - Some more parallelization in deeper code.
- **Documentation**
  - Extensive tutorials for dissimilarities, sampling and the corpus shuffling tool

1.9.7 Version 0.2.0 (2021-07-06) (@lfavre)

- Numerous bug fixes (errors and mismatches with the original java implementation)
- Documentation of slight differences between the java implementation and our implementation that might impact results
- Various minor performance improvements
- Multiprocessing-based parallelization of the costly disorder computation
- Gamma-k and Gamma-cat implementation
- Levenshtein categorical dissimilarity
- New continuum sampler, and an API to create your own continuum sampler
- Addition of the Corpus Shuffling Tool for benchmarking the gamma family of agreement measures
- Rationalized the usage of various data structures to sortedcontainers to prevent any non-deterministic side effects
- Additionnal manipulation and creation methods for continuua
- New options for the command line tool, such as json output or seeding
- New visualization output for alignments in jupyter notebooks
- Changed the mixed integer programming solver from the unprecise ECOS_BB to GLPK_MI or CBC

BIBLIOGRAPHY

[mathet2015] Yann Mathet et Al. The Unified and Holistic Method Gamma () for Inter-Annotator Agreement Measure and Alignment (Yann Mathet, Antoine Widlöcher, Jean-Philippe Métivier)

[mathet2018] Yann Mathet The Agreement Measure Gamma-Cat : a Complement to Gamma Focused on Categorization of a Continuum (Yann Mathet 2018)

[mathet2015] Yann Mathet et Al. The Unified and Holistic Method Gamma () for Inter-Annotator Agreement Measure and Alignment (Yann Mathet, Antoine Widlöcher, Jean-Philippe Métivier)

[mathet2015] Yann Mathet et Al. The Unified and Holistic Method Gamma () for Inter-Annotator Agreement Measure and Alignment (Yann Mathet, Antoine Widlöcher, Jean-Philippe Métivier)

[mathet2018] Yann Mathet The Agreement Measure Gamma-Cat : a Complement to Gamma Focused on Categorization of a Continuum (Yann Mathet 2018)

[mathet2015] Yann Mathet et Al. The Unified and Holistic Method Gamma () for Inter-Annotator Agreement Measure and Alignment (Yann Mathet, Antoine Widlöcher, Jean-Philippe Métivier)

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

U

V