pyndl

Konstantin Sering, Marc Weitz, David-Elias Künstle, Lennart Schn

Nov 24, 2022

CONTENTS:

pyndl implements Naïve Discriminative Learning (NDL) in Python. NDL is an incremental learning algorithm grounded in the principles of discrimination learning and motivated by animal and human learning research. Lately, NDL has become a popular tool in language research to examine large corpora and vocabularies, with 750,000 spoken word tokens and a vocabulary size of 52,402 word types. In contrast to previous implementations, *pyndl* allows for a broader range of analysis, including non-English languages, adds further learning rules and provides better maintainability while having the same fast processing speed. As of today, it supports multiple research groups in their work and led to several scientific publications.

CHAPTER

ONE

QUICKSTART

1.1 Installation

First, you need to install *pyndl*. The easiest way to do this is using pip:

```
pip install --user pyndl
```

Warning: If you are using any other operating system than Linux this process can be more difficult. Check out *Installation* for more detailed installation instruction. However, currently we can only ensure the expected behaviour on Linux system. Be aware that on other operating system some functionality may not work

1.2 Naive Discriminative Learning

Naive Discriminative Learning, henceforth NDL, is an incremental learning algorithm based on the learning rule of Rescorla and Wagner[1], which describes the learning of direct associations between cues and outcomes. The learning is thereby structured in events where each event consists of a set of cues which give hints to outcomes. Outcomes can be seen as the result of an event, where each outcome can be either present or absent. NDL is naive in the sense that cue-outcome associations are estimated separately for each outcome.

The Rescorla-Wagner learning rule describes how the association strength ΔV_i^t at time t changes over time. Time is here described in form of learning events. For each event the association strength is updated as

$$V_i^{t+1} = V_i^t + \Delta V_i^t$$

Thereby, the change in association strength ΔV_i^t is defined as

$$\Delta V_i^t = \begin{cases} 0 & \text{if ABSENT}(C_i, t) \\ \alpha_i \beta_1 \left(\lambda - \sum_{\text{PRESENT}(C_j,t)} V_j\right) & \text{if PRESENT}(C_j, t) \;\&\; \text{PRESENT}(O, t) \\ \alpha_i \beta_2 \left(0 - \sum_{\text{PRESENT}(C_j,t)} V_j\right) & \text{if PRESENT}(C_j, t) \;\&\; \text{ABSENT}(O, t) \end{cases}$$

with

- α_i being the salience of the cue i
- β_1 being the salience of the situation in which the outcome occurs
- β_2 being the salience of the situation in which the outcome does not occur

[1] Rescorla, R. A., & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and non-reinforcement. *Classical conditioning II: Current research and theory*, 2, 64-99.

- λ being the the maximum level of associative strength possible

Note: Usually, the parameters are set to $\alpha_i = \alpha_j \; \forall i, j$, $\beta_1 = \beta_2$ and $\lambda = 1$

1.3 Usage

Analyzing data with *pyndl* involves three steps

1. The data has to be preprocessed into the correct format
2. One of the learning methods of *pyndl* is used to learn the desired associations
3. The learned association (commonly also called weights) can be stored or directly be analyzed further.

In the following, a usage example of *pyndl* is provided, in which the first two of the three steps are described for learning the associations between bigrams and meanings. The first section of this example focuses on the correct preparation of the data with inbuilt methods. However, it is worth to note that the learning algorithm itself does not require the data to be preprocessed by *pyndl*, nor it is limited by that. The `pyndl.preprocess` module should rather be seen as a collection of established and commonly used preprocessing methods within the context of NDL. Custom preprocessing can be used as long as the created event files follow the structure as outlined in the next section. The second section, describes how the associations can be learned using *pyndl*, while the last section describes how this can be exported and, for instance, loaded in R for further investigation.

1.3.1 Data Preparation

To analyse any data using *pyndl* requires them to be in the long format as an utf-8 encoded tab delimited gzipped text file with a header in the first line and two columns:

1. the first column contains an underscore delimited list of all cues
2. the second column contains an underscore delimited list of all outcomes
3. each line therefore represents an event with a pair of a cue and an outcome (occurring one time)
4. the events (lines) are ordered chronologically

The algorithm itself is agnostic to the actual domain as long as the data is tokenized as Unicode character strings. While *pyndl* provides some basic preprocessing for grapheme tokenization (see for instance the following examples), the tokenization of ideograms, pictograms, logograms, and speech has to be implemented manually. However, generic implementations are welcome as a contribution.

Creating Grapheme Clusters From Wide Format Data

Often data which should be analysed is not in the right format to be processed with *pyndl*. To illustrate how to get the data in the right format we use data from Baayen, Milin, Đurđević, Hendrix & Marelli[2] as an example:

[2] Baayen, R. H., Milin, P., Đurđević, D. F., Hendrix, P., & Marelli, M. (2011). An amorphous model for morphological processing in visual comprehension based on naive discriminative learning. *Psychological review*, 118(3), 438.

Table 1			
Word	Frequency	Lexical Meaning	Number
hand	10	HAND	
hands	20	HAND	PLURAL
land	8	LAND	
lands	3	LAND	PLURAL
and	35	AND	
sad	18	SAD	
as	35	AS	
lad	102	LAD	
lads	54	LAD	PLURAL
lass	134	LASS	

Table 1 shows some words, their frequencies of occurrence and their meanings as an artificial lexicon in the wide format. In the following, the letters (unigrams and bigrams) of the words constitute the cues, whereas the meanings represent the outcomes.

As the data in table 1 are artificial we can generate such a file for this example by expanding table 1 randomly regarding the frequency of occurrence of each event. The resulting event file lexample.tab.gz consists of 420 lines (419 = sum of frequencies + 1 header) and looks like the following (nevertheless you are encouraged to take a closer look at this file using any text editor of your choice):

Cues	Outcomes
#h_ha_an_nd_ds_s#	hand_plural
#l_la_ad_d#	lad
#l_la_as_ss_s#	lass

Creating Grapheme Clusters From Corpus Data

Often the corpus which should be analysed is only a raw utf-8 encoded text file that contains huge amounts of text. From here on we will refer to such a file as a corpus file. In the corpus files several documents can be stored with a `---end.of.document---` or `---END.OF.DOCUMENT---` string marking where an old document finished and a new document starts.

The *pyndl.preprocess* module (besides other things) provides the functionality to directly generate an event file based on a raw corpus file and filter it:

```
>>> from pyndl import preprocess
>>> preprocess.create_event_file(corpus_file='docs/data/lcorpus.txt',
...                              event_file='docs/data/levent.tab.gz',
...                              allowed_symbols='a-zA-Z',
...                              context_structure='document',
...                              event_structure='consecutive_words',
...                              event_options=(1, ),
...                              cue_structure='bigrams_to_word')
```

Here we use the example corpus lcorpus.txt to produce an event file `levent.tab.gz` which (uncompressed) looks like this:

Cues	Outcomes
an_#h_ha_d#_nd	hand
ot_fo_oo_#f_t#	foot
ds_s#_an_#h_ha_nd	hands

Note: *pyndl.corpus* allows you to generate such a corpus file from a bunch of gunzipped xml subtitle files filled with words.

1.3.2 Learn the associations

The strength of the associations for the data can now easily be computed using the *pyndl.ndl.ndl* function from the *pyndl.ndl* module:

```
>>> from pyndl import ndl
>>> weights = ndl.ndl(events='docs/data/levent.tab.gz',
...                   alpha=0.1, betas=(0.1, 0.1), method="threading")
```

1.3.3 Save and load a weight matrix

To save time in the future, we recommend saving the weights. For compatibility reasons we recommend saving the weight matrix in the netCDF format[3]:

```
>>> weights.to_netcdf('docs/data/weights.nc')
```

Now, the saved weights can later be reused or be analysed in Python or R. In Python the weights can simply be loaded with the xarray module:

```
>>> import xarray
>>> with xarray.open_dataarray('docs/data/weights.nc') as weights_read:
...     weights_read
```

In R you need the ncdf4 package to load a in netCDF format saved matrix:

```
> #install.packages("ncdf4") # uncomment to install
> library(ncdf4)
> weights_nc <- nc_open(filename = "docs/data/weights.nc")
> weights_read <- t(as.matrix(ncvar_get(nc = weights_nc, varid = "__xarray_dataarray_
↪variable__")))
> rownames(weights_read) <- ncvar_get(nc = weights_nc, varid = "outcomes")
> colnames(weights_read) <- ncvar_get(nc = weights_nc, varid = "cues")
> nc_close(nc = weights_nc)
> rm(weights_nc)
```

[3] Unidata (2012). NetCDF. doi:10.5065/D6H70CW6. Retrieved from http://doi.org/10.5065/D6RN35XM)

1.3.4 Clean up

In order to keep everything clean we might want to remove all the files we created in this tutorial:

```
>>> import os
>>> os.remove('docs/data/levent.tab.gz')
```

Installation

Supported systems and versions

pyndl currently is only tested and mainly used on 64-bit Linux systems. However, it is possible to install it on other operating systems, but be aware that some functionality might not work or will not work as intended. Therefore be extra careful and run the test suite after installing it on a non Linux system.

Note: If you face problems with installing *pyndl* with *pip*, it might be helpful to use Minicoda to install the following dependencies:

```
conda install numpy cython pandas xarray netCDF4 numpydoc pip
```

The reason behind this is that during the installation process of *pyndl* Cython extension need to be installed, if no pre-compiled wheel could be found for your operating system and architecture. To compile Cython extensions some further steps need to be done, which is described in the Cython documentation . These steps depend on your operating system. Installing Cython with *conda install cython* should add all the necessary additional programs and files and no further steps are needed.

Linux

If you want to install *pyndl* on Linux the easiest way is to install it from pypi with:

```
pip install --user pyndl
```

MacOS

If you want to install *pyndl* on MacOS you can also install it from pypi. However, the installation will not have *openmp* support. Sometimes an error is shown during the installation, but the installations succeeds nonetheless. Before filing a bug report please check if you can run the examples from the documentation.

Install *pyndl* with:

```
pip install --user pyndl
```

Windows 10

Note: You might need to enable the `bash` within Windows 10 first to be able to follow the following instructions.

After installing Anaconda or Miniconda, first install the dependencies with the `conda` command in the bash or the Miniconda terminal:

```
conda update conda
conda install numpy cython pandas xarray netCDF4 numpydoc pip
```

After the installation of the dependencies finished successfully you should be able to install `pyndl` with pip:

```
pip install --user pyndl
```

Warning: This procedure is experimental and might not work. As long as we do not actively support Windows 10 be aware that these installation instructions can fail or the installed package does not always works as intended!

Background

Naive Discriminative Learning

Terminology

Before explaining Naive Discriminative Learning (NDL) in detail, we want to give you a brief overview over important notions:

cue :
A cue is something that gives a hint on something else. The something else is called outcome. Examples for cues in a text corpus are trigraphs or preceding words for the word or meaning of the word.

outcome :
The outcome is the result of an event. Examples are words, the meaning of the word, or lexomes.

event :
An event connects cues with outcomes. In any event one or more unordered cues are present and one or more outcomes are present.

weights :
The weights represent the learned experience / association between all cues and outcomes of interest. Usually, some meta data is stored alongside the learned weights.

Rescorla Wagner learning rule

In order to update the association strengths (weights) between cues and outcomes we do for each event the following:

We calculate the activation (prediction) a_j for each outcome o_j by using all present cues C_{PRESENT}:

$$a_j = \sum_{i \text{ for } c_i \in C_{\text{PRESENT}}} w_{ij}$$

After that, we calculate the update Δw_{ij} for every cue-outcome-combination:

$$\Delta w_{ij} \begin{cases} 0 & \text{if cue } c_i \text{ is absent} \\ \alpha_i \beta_1 \cdot (\lambda - a_j) & \text{if outcome } o_j \text{ and cue } c_i \text{ is present.} \\ \alpha_i \beta_2 \cdot (0 - a_j) & \text{if outcome } o_j \text{ is absent and cue } c_i \text{ is present.} \end{cases}$$

In the end, we update all weights according to $w_{ij} = w_{ij} + \Delta w_{ij}$.

Note: If we set all the α's and β's to a fixed value we can replace them in the equation with a general learning parameter $\eta = \alpha \cdot \beta$.

In matrix notation

We can rewrite the Rescorla-Wagner learning rule into matrix notation with a binary cue (input) vector $\vec{c}$, which is one for each cue present in the event and zero for all other cues. Respectively, we define a binary outcome (output) vector $\vec{o}$, which is one for each outcome present in the event and zero if the outcome is not present. In order to stick close to the definition above we can define the activation vector as $\vec{a} = W^T \vec{c}$. Here W^T denotes the transposed matrix of the weight matrix W.

For simplicity let us assume we have a fixed learning rate $\eta = \alpha\beta$. We will relax this simplification in the end. We can rewrite the above rule as:

$$\begin{aligned} \Delta &= \eta \vec{c} \cdot (\lambda \vec{o} - \vec{a})^T \\ &= \eta \vec{c} \cdot (\lambda \vec{o} - W^T \cdot \vec{c})^T \end{aligned}$$

Let us first check the dimensionality of the matrices:

Δ is the update of the weight matrix W and therefore needs to have the same dimensions $n \times m$ where n denotes the number of cues (inputs) and m denotes the number of outcomes (outputs).

The cue vector $\vec{c}$ can be seen as a matrix with dimensions $n \times 1$ and the outcome vector can be seen as a matrix with dimensions $m \times 1$. Let us tabulate the dimensions:

$\lambda \vec{o}$	$m \times 1$
W^T	$m \times n$
$\vec{c}$	$n \times 1$
$W^T \cdot \vec{c}$	$m \times 1 = (m \times n) \cdot (n \times 1)$
$\lambda \vec{o} - W^T \cdot \vec{c}$	$m \times 1 = (m \times 1) - (m \times 1)$
$(\lambda \vec{o} - W^T \cdot \vec{c})^T$	$1 \times m = (m \times 1)^T$
$\eta \vec{c} \cdot (\lambda \vec{o} - W^T \cdot \vec{c})$	$n \times m = (n \times 1) \cdot (1 \times n)$

We therefore end with the right set of dimensions. We now can try to simplify / rewrite the equation.

$$\begin{aligned} \Delta &= \eta \vec{c} \cdot ((\lambda \vec{o})^T - (W^T \cdot \vec{c})^T) \\ &= \eta \vec{c} \cdot (\lambda \vec{o}^T - \vec{c}^T \cdot W) \\ &= \eta \lambda \vec{c} \cdot \vec{o}^T - \eta \vec{c} \cdot \vec{c}^T \cdot W \end{aligned}$$

If we now look at the full update:

$$\begin{aligned} W_{t+1} &= W_t + \Delta_t \\ &= W + \Delta \\ &= W + \eta\lambda\vec{c} \cdot \vec{o}^T - \eta\vec{c} \cdot \vec{c}^T \cdot W \\ &= \eta\lambda\vec{c} \cdot \vec{o}^T + W - \eta\vec{c} \cdot \vec{c}^T \cdot W \\ &= \eta\lambda\vec{c} \cdot \vec{o}^T + (1 - \eta\vec{c} \cdot \vec{c}^T) \cdot W \end{aligned}$$

We therefore see that the Rescorla-Wagner update is an affine (linear) transformation[1] in the weights W with an intercept of $\eta\lambda\vec{c} \cdot \vec{o}^T$ and a slope of $(1 - \eta\vec{c} \cdot \vec{c}^T)$.

In index notation we can write:

$$\begin{aligned} W^{t+1} &= W^t + \eta\vec{c} \cdot (\lambda\vec{o}^T - \vec{c}^T \cdot W) \\ W_{ij}^{t+1} &= W_{ij}^t + \eta c_i(\lambda o_j - \sum_k c_k W_{kj}) \end{aligned}$$

Note: Properties of the transpose[4] with A and B matrices and α skalar:

$$(A^T)^T = A$$

$$(A + B)^T = A^T + B^T$$

$$(\alpha A)^T = \alpha A^T$$

$$(A \cdot B)^T = B^T \cdot A^T$$

Other Learning Algorithms

The delta rule[2] is a gradient descent learning rule for updating the weights of the inputs to artificial neurons in a single-layer neural network. It is a special case of the more general backpropagation algorithm[3].

The delta rule can be expressed as:

$$\Delta_{ij} = \alpha(t_j - y_j)\partial_{h_j} g(h_j) x_i$$

In the terminology above we can identify the actual output with $y_j = g(h_j) = g\left(\sum_i w_{ij} c_i\right)$, the cues with $x_i = c_i$, under the assumption that o_j is binary (i. e. either zero or one) we can write $t_j = \lambda o_j$, the learning rate $\alpha = \eta = \alpha\beta$. Substituting this equalities results in:

$$\Delta_{ij} = \eta(\lambda o_j - g\left(\sum_i w_{ij} c_i\right))\partial_{h_j} g(h_j) c_i$$

In order to end with the Rescorla-Wagner learning rule we need to set the neuron's activation function $g(h_j)$ to the identity function, i. e. $g(h_j) = 1 \cdot h_j + 0 = h_j = \sum_i w_{ij} c_i$. The derivative in respect to h_j is $\partial_{h_j} g(h_j) = 1$ for any input h_j.

[1] https://en.wikipedia.org/wiki/Affine_transformation
[4] https://en.wikipedia.org/wiki/Transpose
[2] https://en.wikipedia.org/wiki/Delta_rule
[3] https://en.wikipedia.org/wiki/Backpropagation

We now have:

$$\begin{aligned}\Delta_{ij} &= \eta(\lambda o_j - \sum_i w_{ij} c_i) \cdot 1 \cdot c_i \\ &= \eta(\lambda o_j - \sum_i w_{ij} c_i) c_i \\ &= \eta c_i (\lambda o_j - \sum_i w_{ij} c_i)\end{aligned}$$

Assuming the cue vector is binary the vector c_i effectively filters those updates of the present cues and sets all updates of the cues that are not present to zero. Additionally, we can rewrite the equation above into vector notation (without indices):

$$\begin{aligned}\Delta_{ij} &= \eta c_i (\lambda o_j - \sum_i w_{ij} c_i) \\ &= \eta c_i (\lambda o_j - \sum_i w_{ij} c_i)\end{aligned}$$

$$\Delta = \eta \vec{c} \cdot (\lambda \vec{o}^T - W^T \cdot \vec{c})^T$$

This is exactly the form of the Rescorla-Wagner rule rewritten in matrix notation.

Conclusion

In conclusion, the Rescorla-Wagner learning rule, which only allows for one α and one β and therefore one learning rate $\eta = \alpha\beta$ is exactly the same as a single layer backpropagation gradient decent method (the delta rule) where the neuron's activation function $g(h_j)$ is set to the identity $g(h_j) = h_j$ and the inputs $x_i = c_i$ and target outputs $t_j = \lambda o_j$ to be binary.

References

Usage Examples

Lexical example

The lexical example illustrates the Rescorla-Wagner equations[1]. This example is taken from Baayen, Milin, Đurđević, Hendrix and Marelli[2].

Premises

1. Cues are associated with outcomes and both can be present or absent
2. Cues are segment (letter) unigrams, bigrams, ...
3. Outcomes are meanings (word meanings, inflectional meanings, affixal meanings), ...
4. PRESENT(X, t) denotes the presence of cue or outcome X at time t
5. ABSENT(X, t) denotes the absence of cue or outcome X at time t

[1] Rescorla, R. A., & Wagner, A. R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and non-reinforcement. *Classical conditioning II: Current research and theory*, 2, 64-99.

[2] Baayen, R. H., Milin, P., Đurđević, D. F., Hendrix, P., & Marelli, M. (2011). An amorphous model for morphological processing in visual comprehension based on naive discriminative learning. *Psychological review*, 118(3), 438.

6. The association strength V_i^{t+1} of cue C_i with outcome O at time $t+1$ is defined as $V_i^{t+1} = V_i^t + \Delta V_i^t$
7. The change in association strength ΔV_i^t is defined as in (1.1) with
   - α_i being the salience of the cue i
   - β_1 being the salience of the situation in which the outcome occurs
   - β_2 being the salience of the situation in which the outcome does not occur
   - λ being the the maximum level of associative strength possible
8. Default settings for the parameters are: $\alpha_i = \alpha_j \; \forall i, j$, $\beta_1 = \beta_2$ and $\lambda = 1$

$$\Delta V_i^t = \begin{cases} 0 & \text{if ABSENT}(C_i, t) \\ \alpha_i \beta_1 \left(\lambda - \sum_{\text{PRESENT}(C_j,t)} V_j\right) & \text{if PRESENT}(C_j, t) \;\&\; \text{PRESENT}(O, t) \\ \alpha_i \beta_2 \left(0 - \sum_{\text{PRESENT}(C_j,t)} V_j\right) & \text{if PRESENT}(C_j, t) \;\&\; \text{ABSENT}(O, t) \end{cases} \tag{1.1}$$

See comparison_of_algorithms for alternative formulations of the Rescorla Wagner learning rule.

Data

Table 1			
Word	Frequency	Lexical Meaning	Number
hand	10	HAND	
hands	20	HAND	PLURAL
land	8	LAND	
lands	3	LAND	PLURAL
and	35	AND	
sad	18	SAD	
as	35	AS	
lad	102	LAD	
lads	54	LAD	PLURAL
lass	134	LASS	

Table 1 shows some words, their frequencies of occurrence and their meanings as an artificial lexicon in the wide format. In the following, the letters (unigrams and bigrams) of the words constitute the cues, the meanings represent the outcomes.

Analyzing any data using *pyndl* requires them to be in the long format as an utf-8 encoded tab delimited gzipped text file with a header in the first line and two columns:

1. the first column contains an underscore delimited list of all cues
2. the second column contains an underscore delimited list of all outcomes
3. each line therefore represents an event with a pair of a cue and an outcome (occurring one time)
4. the events (lines) are ordered chronologically

As the data in table 1 are artificial we can generate such a file for this example by expanding table 1 randomly regarding the frequency of occurrence of each event. The resulting event file lexample.tab.gz consists of 420 lines (419 = sum of frequencies + 1 header) and looks like the following (nevertheless you are encouraged to take a closer look at this file using any text editor of your choice):

Cues	Outcomes
#h_ha_an_nd_ds_s#	hand_plural
#l_la_ad_d#	lad
#l_la_as_ss_s#	lass

pyndl.ndl module

We can now compute the strength of associations (weights or weight matrix) after the presentation of the 419 tokens of the 10 words using *pyndl.ndl*. *pyndl.ndl* provides the two functions *pyndl.ndl.ndl* and *pyndl.ndl.dict_ndl* to calculate the weights for all outcomes over all events. *pyndl.ndl.ndl* itself provides to methods regarding estimation, `openmp` and `threading`. We have to specify the path of our event file lexample.tab.gz and for this example set $\alpha = 0.1$, $\beta_1 = 0.1$, $\beta_2 = 0.1$ with leaving $\lambda = 1.0$ at its default value. You can use *pyndl* directly in a Python3 Shell or you can write an executable script, this is up to you. For educational purposes we use a Python3 Shell in this example.

pyndl.ndl.ndl

pyndl.ndl.ndl is a parallel Python implementation using numpy, multithreading and a binary format which is created automatically. It allows you to choose between the two methods `openmp` and `threading`, with the former one using openMP and therefore being expected to be faster when analyzing larger data. Unfortunately, `openmp` is only available on Linux right now, therefore all examples use `threading`. Besides, you can set three technical arguments which we will not change here:

1. `n_jobs` (int) giving the number of threads in which the job should be executed (default=2)
2. `sequence` (int) giving the length of sublists generated from all outcomes (default=10)
3. `remove_duplicates` (logical) to make cues and outcomes unique (default=None; which will raise an error if the same cue is present multiple times in the same event)

Let's start:

```
>>> from pyndl import ndl
>>> weights = ndl.ndl(events='docs/data/lexample.tab.gz', alpha=0.1,
...                   betas=(0.1, 0.1), method='threading')
>>> weights
<xarray.DataArray (outcomes: 8, cues: 15)>
...
```

`weights` is an `xarray.DataArray` of dimension `len(outcomes)`, `len(cues)`. Our unique, chronologically ordered outcomes are 'hand', 'plural', 'lass', 'lad', 'land', 'as', 'sad', 'and'. Our unique, chronologically ordered cues are '#h', 'ha', 'an', 'nd', 'ds', 's#', '#l', 'la', 'as', 'ss', 'ad', 'd#', '#a', '#s', 'sa'. Therefore all three indexing methods

```
>>> weights[1, 5]
<xarray.DataArray ()>
...
>>> weights.loc[{'outcomes': 'plural', 'cues': 's#'}]
<xarray.DataArray ()>
array(0.076988...)
Coordinates:
    outcomes  <U6 'plural'
    cues      <U2 's#'
...
```

(continues on next page)

(continued from previous page)

```
>>> weights.loc['plural'].loc['s#']
<xarray.DataArray ()>
array(0.076988...)
Coordinates:
    outcomes  <U6 'plural'
    cues      <U2 's#'
...
```

return the weight of the cue 's#' (the unigram 's' being the word-final) for the outcome 'plural' (remember counting in Python does start at 0) as ca. 0.077 and hence indicate 's#' being a marker for plurality.

pyndl.ndl.ndl also allows you to continue learning from a previous weight matrix by specifying the `weight` argument:

```
>>> weights2 = ndl.ndl(events='docs/data/lexample.tab.gz', alpha=0.1,
...                    betas=(0.1, 0.1), method='threading', weights=weights)
>>> weights2
<xarray.DataArray (outcomes: 8, cues: 15)>
array([[ 0.24...
...
...]])
Coordinates:
  * outcomes  (outcomes) <U6 'hand' 'plural'...
  * cues      (cues) <U2 '#h' 'ha' 'an' 'nd'...
Attributes:...
    date:...
    event_path:...
...
```

As you may have noticed already, *pyndl.ndl.ndl* provides you with meta data organized in a `dict` which was collected during your calculations. Each entry of each `list` of this meta data therefore references one specific moment of your calculations:

```
>>> print('Attributes: ' + str(weights2.attrs))
Attributes: ...
```

pyndl.ndl.dict_ndl

pyndl.ndl.dict_ndl is a pure Python implementation, however, it differs from *pyndl.ndl.ndl* regarding the following:

1. there are only two technical arguments: `remove_duplicates` (logical) and `make_data_array` (logical)
2. by default, no longer an `xarray.DataArray` is returned but a `dict` of dicts
3. however, you are still able to get an `xarray.DataArray` by setting `make_data_array=True`
4. the case $\alpha_i \neq \alpha_j$ can be handled by specifying a `dict` consisting of the cues as keys and corresponding α's

Therefore

```
>>> weights = ndl.dict_ndl(events='docs/data/lexample.tab.gz',
...                        alphas=0.1, betas=(0.1, 0.1))
```

(continues on next page)

(continued from previous page)

```
>>> weights['plural']['s#'] # doctes: +ELLIPSIS
0.076988227...
```

yields approximately the same results as before, however, you now can specify different α's per cue and as in *pyndl.ndl.ndl* continue learning or do both:

```
>>> alphas_cues = dict(zip(['#h', 'ha', 'an', 'nd', 'ds', 's#', '#l', 'la', 'as', 'ss',
↪'ad', 'd#', '#a', '#s', 'sa'],
...                        [0.1, 0.2, 0.3, 0.4, 0.1, 0.2, 0.3, 0.1, 0.2, 0.1, 0.2, 0.1,␣
↪0.3, 0.1, 0.2]))
>>> weights = ndl.dict_ndl(events='docs/data/lexample.tab.gz',
...                        alphas=alphas_cues, betas=(0.1, 0.1))
>>> weights2 = ndl.dict_ndl(events='docs/data/lexample.tab.gz',
...                         alphas=alphas_cues, betas=(0.1, 0.1),
...                         weights=weights)
```

If you prefer to get a `xarray.DataArray` returned you can set the flag `make_data_array=True`:

```
>>> weights = ndl.dict_ndl(events='docs/data/lexample.tab.gz',
...                        alphas=alphas_cues, betas=(0.1, 0.1),
...                        make_data_array=True)
>>> weights
<xarray.DataArray (outcomes: 8, cues: 15)>
...
```

A minimal workflow example

As you should have a basic understanding of *pyndl.ndl* by now, the following example will show you how to:

1. generate an event file based on a raw corpus file
2. count cues and outcomes
3. filter the events
4. learn the weights as already shown in the lexical learning example
5. save and load a weight matrix (netCDF format)
6. load a weight matrix (netCDF format) into R for further analyses

Generate an event file based on a raw corpus file

Suppose you have a raw utf-8 encoded corpus file (by the way, *pyndl.corpus* allows you to generate such a corpus file from a bunch of gunzipped xml subtitle files filled with words, which we will not cover here). For example take a look at *lcorpus.txt*_.

To analyse the data, you need to convert the file into an event file similar to lexample.tab.gz in our lexical learning example, as currently there is only one word per line and neither is there the column for cues nor for outcomes:

```
hand
foot
hands
```

The *pyndl.preprocess* module (besides other things) allows you to generate an event file based on a raw corpus file and filter it:

```
>>> import pyndl
>>> from pyndl import preprocess
>>> preprocess.create_event_file(corpus_file='docs/data/lcorpus.txt',
...                              event_file='docs/data/levent.tab.gz',
...                              allowed_symbols='a-zA-Z',
...                              context_structure='document',
...                              event_structure='consecutive_words',
...                              event_options=(1, ),
...                              cue_structure='bigrams_to_word')
```

The function *pyndl.preprocess.create_event_file* has several arguments which you might have to change to suit them your data, so you are strongly recommended to read its documentation. We set `context_structure='document'` as in this case the context is the whole document, `event_structure='consecutive_words'` as these are our events, `event_options=(1, )` as we define an event to be one word and `cue_structure='bigrams_to_word'` to set cues being bigrams. There are also several technical arguments you can specify, which we will not change here. Our generated event file `levent.tab.gz` now looks (uncompressed) like this:

Cues	Outcomes
an_#h_ha_d#_nd	hand
ot_fo_oo_#f_t#	foot
ds_s#_an_#h_ha_nd	hands

Count cues and outcomes

We can now count the cues and outcomes in our event file using the *pyndl.count* module and also generate id maps for cues and outcomes:

```
>>> from pyndl import count
>>> freq, cue_freq_map, outcome_freq_map = count.cues_outcomes(event_file_name='docs/
↪data/levent.tab.gz')
>>> freq
12
>>> cue_freq_map
Counter({...})
>>> outcome_freq_map
Counter({...})
>>> cues = list(cue_freq_map.keys())
>>> cues.sort()
>>> cue_id_map = {cue: ii for ii, cue in enumerate(cues)}
>>> cue_id_map
{...}
>>> outcomes = list(outcome_freq_map.keys())
>>> outcomes.sort()
>>> outcome_id_map = {outcome: nn for nn, outcome in enumerate(outcomes)}
>>> outcome_id_map
{...}
```

Filter the events

As we do not want to include the outcomes 'foot' and 'feet' in this example as well as their cues '#f', 'fo' 'oo', 'ot', 't#', 'fe', 'ee' 'et', we use the *pyndl.preprocess* module again, filtering our event file and update the id maps for cues and outcomes:

```
>>> preprocess.filter_event_file(input_event_file='docs/data/levent.tab.gz',
...                              output_event_file='docs/data/levent.tab.gz.filtered',
...                              remove_cues=('#f', 'fo', 'oo', 'ot', 't#', 'fe', 'ee',
↪'et'),
...                              remove_outcomes=('foot', 'feet'))
>>> freq, cue_freq_map, outcome_freq_map = count.cues_outcomes(event_file_name='docs/
↪data/levent.tab.gz.filtered')
>>> cues = list(cue_freq_map.keys())
>>> cues.sort()
>>> cue_id_map = {cue: ii for ii, cue in enumerate(cues)}
>>> cue_id_map
{...}
>>> outcomes = list(outcome_freq_map.keys())
>>> outcomes.sort()
>>> outcome_id_map = {outcome: nn for nn, outcome in enumerate(outcomes)}
>>> outcome_id_map
{...}
```

Alternatively, using *pyndl.preprocess.filter_event_file* you can also specify which cues and outcomes to keep (`keep_cues` and `keep_outcomes`) or remap cues and outcomes (`cue_map` and `outcomes_map`). Besides, there are also some technical arguments you can specify, which will not discuss here.

Last but not least *pyndl.preprocess* does provide some other very useful functions regarding preprocessing of which we did not make any use here, so make sure to go through its documentation.

Learn the weights

Computing the strength of associations for the data is now easy, using for example *pyndl.ndl.ndl* from the *pyndl.ndl* module like in the lexical learning example:

```
>>> from pyndl import ndl
>>> weights = ndl.ndl(events='docs/data/levent.tab.gz.filtered',
...                   alpha=0.1, betas=(0.1, 0.1), method="threading")
```

Save and load a weight matrix

is straight forward using the netCDF format[3]

```
>>> import xarray
>>> weights.to_netcdf('docs/data/weights.nc')
>>> with xarray.open_dataarray('docs/data/weights.nc') as weights_read:
...     weights_read
```

In order to keep everything clean we might want to remove all the files we created in this tutorial:

[3] Unidata (2012). NetCDF. doi:10.5065/D6H70CW6. Retrieved from http://doi.org/10.5065/D6RN35XM)

```
>>> import os
>>> os.remove('docs/data/levent.tab.gz')
>>> os.remove('docs/data/levent.tab.gz.filtered')
```

Widrow-Hoff (WH) learning

There is a Widrow-Hoff learning module called *wh* now in *pyndl*, which uses the same event files and nearly the same function parameters as the *ndl.ndl* function. The main function to call is *wh.wh*. Compared to *ndl.ndl* the *wh.wh* function adds two look-up tables, one for cues and one for outcomes, to its keyword arguments. Each of this look-up tables maps each cue and / or outcome in your event file to a vector. This look-up table has to be an instance *xarray.DataArray* and is passed with the keyword argument *cue_vectors* or *outcome_vectors*. The second dimension of the look-up table needs to be named *cue_vector_dimensions* and *outcome_vector_dimensions* respectively. For more information have a look at the function doc string.

WH example

This example shows that WH learning mimics RW learning, if the cue and outcome vectors are containing unit vectors. Note that WH learning in contrast to the RW learning only has one learning parameter, which is called *eta*. The assumption is that *beta1* equals *beta2*.

```
>>> from pyndl import wh, ndl
>>> import xarray as xr
>>> import numpy as np
>>> events = 'docs/data/event_file_wh.tab.gz'
>>> eta = 0.01  # learning rate
>>> cue_vectors = xr.DataArray(np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]], dtype=float),
...                            dims=('cues', 'cue_vector_dimensions'),
...                            coords={'cues': ['a', 'b', 'c'], 'cue_vector_dimensions':␣
↪['dim1', 'dim2', 'dim3']})
>>> outcome_vectors = xr.DataArray(np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0],␣
↪[0, 0, 0, 1]], dtype=float),
...                                dims=('outcomes', 'outcome_vector_dimensions'),
...                                coords={'outcomes': ['A', 'B', 'C', 'D'],
...                                        'outcome_vector_dimensions': ['dim1', 'dim2',
↪'dim3', 'dim4']})
>>> weights_wh = wh.wh(events, eta, cue_vectors=cue_vectors, outcome_vectors=outcome_
↪vectors, method='numpy')
>>> weights_ndl = ndl.ndl(events, alpha=1.0, betas=(eta, eta), method='threading')
```

The weights returned by *wh.wh* have dimensions *outcome_vector_dimensions* and *cue_vector_dimensions*. Therefore, a direct comparison is not possible. But as the vectors used are unit vectors the first *cue_vector_dimension "dim1"* corresponds to the first cue *"a"* and the second vector dimension corresponds to the second cue etc. If the dimensions are ordered by their names, the equality gets apparent.

```
>>> weights_wh = weights_wh.loc[{'outcome_vector_dimensions': ['dim1', 'dim2', 'dim3',
↪'dim4'],
...                           'cue_vector_dimensions': ['dim1', 'dim2', 'dim3']}]
>>> weights_ndl = weights_ndl.loc[{'outcomes': ['A', 'B', 'C', 'D'], 'cues': ['a', 'b',
↪'c']}]
>>> print(weights_wh)
<xarray.DataArray (outcome_vector_dimensions: 4, cue_vector_dimensions: 3)>
```

(continues on next page)

(continued from previous page)

```
array([[0.06706..., 0.        , 0.        ],
       [0.        , 0.03940..., 0.        ],
       [0.0094... , 0.        , 0.03940...],
       [0.01      , 0.        , 0.        ]])
Coordinates:
  * outcome_vector_dimensions  (outcome_vector_dimensions) <U4 'dim1' ... 'dim4'
  * cue_vector_dimensions      (cue_vector_dimensions) <U4 'dim1' 'dim2' 'dim3'
    outcomes                   <U1 'A'
    cues                       <U1 'a'
Attributes: (12/15)
...
>>> print(weights_ndl)
<xarray.DataArray (outcomes: 4, cues: 3)>
array([[0.06706..., 0.        , 0.        ],
       [0.        , 0.03940..., 0.        ],
       [0.0094... , 0.        , 0.03940...],
       [0.01      , 0.        , 0.        ]])
Coordinates:
  * outcomes  (outcomes) <U1 'A' 'B' 'C' 'D'
  * cues      (cues) <U1 'a' 'b' 'c'
Attributes: (12/17)
...
```

Furthermore, it is possible to only use either *cue_vectors* or *outcome_vectors*. This functionality is Linux only at the moment.

```
>>> weights_wh_cv_only = wh.wh(events, eta, cue_vectors=cue_vectors, method='openmp')
>>> weights_wh_ov_only = wh.wh(events, eta, outcome_vectors=outcome_vectors, method=
↪'openmp')
```

For this example the content of the resulting weights matches the content of the *weights_wh* and *weights_ndl*.

Load a weight matrix to R[4]

We can load a in netCDF format saved matrix into R:

```
> #install.packages("ncdf4") # uncomment to install
> library(ncdf4)
> weights_nc <- nc_open(filename = "docs/data/weights.nc")
> weights_read <- t(as.matrix(ncvar_get(nc = weights_nc, varid = "__xarray_dataarray_
↪variable__")))
> rownames(weights_read) <- ncvar_get(nc = weights_nc, varid = "outcomes")
> colnames(weights_read) <- ncvar_get(nc = weights_nc, varid = "cues")
> nc_close(nc = weights_nc)
> rm(weights_nc)
```

[4] R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Benchmark

Here we compare the performance of Naive Discrimination Learning in *pyndl* with other implementations, namely from the *R* packages *ndl* and *ndl2*.

This document summarizes the benchmarking procedure and results; the code to run these is available in the repository's benchmark folder.

Creating the event files

The benchmark datasets are trigram-to-word events on the text of the German *Graphem* Wikipedia page, that can be generated with `python preprocessing.py`.

```
Cues: ['aph', 'em#', 'hem', 'gra', 'rap', '#gr', 'phe'], Outcome: ['graphem']
Cues: ['aus', '#au', 'us#'], Outcome: ['aus']
Cues: ['wik', '#wi', 'kip', 'iki', 'edi', 'ia#', 'ped', 'ipe', 'dia'], Outcome: [
↪'wikipedia']
...
Cues: ['', '#', '', '#', '', '', ''], Outcome: ['']
Cues: ['#', '', '', '#', '', '', ''], Outcome: ['']
Cues: ['#', '', '#'], Outcome: ['']
...
```

The text contains some non-ASCII characters from various languages. While this is not a problem with *pyndl*, these characters would crash *ndl2*. This is why create cue-outcome events with a basic preprocessing, including just ASCII characters and letters. Please note, that in research applications we recommend using dedicated software for more careful preprocessing.

```
preprocess.create_event_file(corpus_file=txt_file,
                             event_file=ascii_event_file,
                             allowed_symbols='a-zA-Z0-9',
                             context_structure='document',
                             event_structure='consecutive_words',
                             event_options=(1,),  # number of words,
                             cue_structure="trigrams_to_word",
                             lower_case=True,
                             remove_duplicates=True)
```

Based on this event file, we construct large scale event file by concatenating all events multiple times.

```
ascii_events = list(io.events_from_file(ascii_event_file))
io.events_to_file(ascii_events * times, event_dir / f"{times}_times_{ascii_event_file.
↪name}", compatible=True)
```

Running pyndl

Our benchmark compares the runtime of *pyndl*'s NDL implementations on event files of varying size. The naive *dict_ndl* learner is orders of magnitude slower and thus excluded from the comparison. In addition to single processing, both parallel processing methods (*threading* and *OpenMP*) are included. Run the benchmark with `python run_ndl.py`, which saves the wall-clock times in a CSV files.

```
def clock(func, args, **kwargs):
    gc.collect()
    start = time.time()
    result = func(*args, **kwargs)
    stop = time.time()
    duration = stop - start
    return result, duration
...
for r in range(repeats):
    for file_path in event_dir.glob('*.tab.gz'):
        _, duration_omp1 = clock(ndl.ndl, (file_path, ALPHA, BETAS, LAMBDA_), n_jobs=1,␣
↪method='openmp')
        _, duration_omp4 = clock(ndl.ndl, (file_path, ALPHA, BETAS, LAMBDA_), n_jobs=4,␣
↪method='openmp')
        ...
```

Running R *ndl* and *ndl2*

The R benchmark includes the equilibrium learner of *ndl* and the iterative learner of *ndl2*. `R -f run_ndl.R` runs the *ndl2* on the same event files as *pyndl*, while *ndl* only runs on the smaller files because it is orders of magnitude slower.

ndl can be installed from CRAN by running `install.packages('ndl')` in R, while *ndl2* needs to be downloaded from Github first (ndl2, install with `R CMD INSTALL ndl2.tar.gz`).

Both *ndl* and *ndl2* cannot handle compressed event files right away. For *ndl*, we load the full event file into memory (included in time measurement), while *ndl2* reads an uncompressed event files.

```
...
for (r in 1:10) {
    file_path <- event_files[i]
        for (i in 1:n) {
                st_proc_time <- system.time({
                        learner <- learnWeightsTabular(gsub("[.]gz$", "", file_path),␣
↪alpha=0.1, beta=0.1, lambda=1.0, numThreads=1, useExistingFiles=FALSE)
                })
            mt_proc_time <- system.time({
                        learner <- learnWeightsTabular(gsub("[.]gz$", "", file_path),␣
↪alpha=0.1, beta=0.1, lambda=1.0, numThreads=4, useExistingFiles=FALSE)
                })
        ...

                if (nrow(event_df) < 100000) {
                proc_time <- system.time({
                    event_df <- read.delim(gzfile(file_path))
                    output <- estimateWeights(cuesOutcomes=event_df)
                }, gcFirst = TRUE)
```

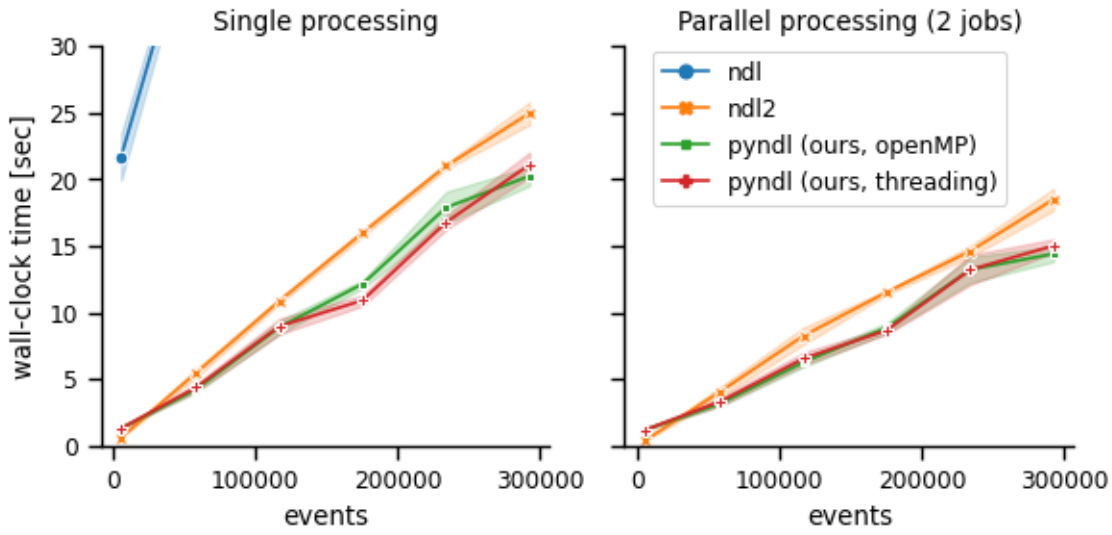
(continues on next page)

(continued from previous page)

```
                }
                ...
        }
        ...
}
```

Results and discussion

We visualize the wall-clock time for the various NDL implementations and number of events as a line plot with error bars for the standard-error within the ten repetitions (`python plot_result.py`, requires the plotting packages *seaborn*). The shown wall-clock times were recorded on a laptop (*Intel(R) Core(TM) i7-8565U CPU* running *Ubuntu 20.4*, *R 3.6* and *python 3.9*).



For small event files the *pyndl* are less than one second slower than *ndl2* but still almost twenty times faster than *ndl*. With increasing number of events, *pyndl* becomes the fastest method in both single and parallel processing (2 jobs).

pyndl processes the event file into a faster accessible format, which results in the overhead for event files. This is similarly done in *ndl2* where this overhead for small event files seems to be less time consuming. However, in contrast to the implementations in *ndl2* and *ndl*, the implementation in *pyndl* never reads the full event file into memory, which is faster and has a smaller memory footprint than its competitors.

Tips & Tricks

This is collection of more or less unrelated tips and tricks that can be helpful during development and maintanance.

Running `pyndl` within `R` code

In order to run `pyndl` within `R` code first install Python and `pyndl` as described in the install instructions. Make sure `pyndl` runs for your user within Python.

Now we can switch to `R` and install the `reticulate` package (https://cran.r-project.org/web/packages/reticulate/vignettes/introduction.html) After having the `reticulate` package installed we can run within R the following code:

```
library(reticulate)

learn_weights <- function(event_file) {
    py_env <- py_run_string(
        paste(
              "from pyndl import ndl",
              paste0("weights = ndl.ndl('", event_file, "', alpha=0.01, betas=(1.0, 1.0),
↪ remove_duplicates=True)"),
              "weight_matrix = weights.data",
              "outcome_names = weights.coords['outcomes'].values",
              "cue_names = weights.coords['cues'].values",
              sep = "\n"
        ),
        convert = FALSE
    )
    wm <- py_to_r(py_env$weight_matrix)
    rownames(wm) <- py_to_r(py_env$outcome_names)
    colnames(wm) <- py_to_r(py_env$cue_names)
    py_run_string(
        paste(
              "del cue_names",
              "del outcome_names",
              "del weight_matrix",
              "del weights",
              sep = "\n"
        ),
        convert = FALSE
    )
    wm
}
```

After having defined this funtion a gzipped tab seperated event file can be learned using:

```
wm <- learn_weights('event_file.tab.gz')
```

Note that this code needs at the moment slightly more than two times the size of the weights matrix.

There might be a way to learn the weight matrix without any copying between R and Python, but this needs to be elaborated a bit further. The basic idea is

1. to create the the matrix in R (in Fortran mode),
2. borrow / make the matrix available in Python,

3. transpose the matrix in Python to get it into C mode
4. learn the weights in place,
5. Check that the matrix in R has the weights learned as a side effect of the Python code.

Further reading:

- https://cran.r-project.org/web/packages/reticulate/vignettes/introduction.html
- https://cran.r-project.org/web/packages/reticulate/vignettes/arrays.html
- https://stackoverflow.com/questions/44379525/r-reticulate-how-do-i-clear-a-python-object-from-memory

API Documentation

pyndl.activation

pyndl.activation provides the functionality to estimate activation of a trained ndl model for given events. The trained ndl model is thereby represented as the outcome-cue weights.

pyndl.activation.**activation**(*events*, *weights*, ***, *n_jobs=1*, *number_of_threads=None*, *remove_duplicates=None*, *ignore_missing_cues=False*)

Estimate activations for given events in event file and outcome-cue weights.

Memory overhead for multiprocessing is one copy of weights plus a copy of cues for each thread.

Parameters

events
[generator or str] generates cues, outcomes pairs or the path to the event file

weights
[xarray.DataArray or dict[dict[float]]] the xarray.DataArray needs to have the dimensions 'outcomes' and 'cues' the dictionaries hold weight[outcome][cue].

n_jobs
[int] a integer giving the number of threads in which the job should executed

remove_duplicates
[{None, True, False}] if None raise a ValueError when the same cue is present multiple times in the same event; True make cues unique per event; False keep multiple instances of the same cue (this is usually not preferred!)

ignore_missing_cues
[{True, False}] if True function ignores cues which are in the test dataset but not in the weight matrix if False raises a KeyError for cues which are not in the weight matrix

Returns

activations
[xarray.DataArray] with dimensions 'outcomes' and 'events'. Contains coords for the outcomes. returned if weights is instance of xarray.DataArray

or
activations
[dict of numpy.arrays] the first dict has outcomes as keys and dicts as values the list has a activation value per event returned if weights is instance of dict

pyndl.corpus

pyndl.corpus generates a corpus file (outfile) out of a bunch of gunzipped xml subtitle files in a directory and all its subdirectories.

class pyndl.corpus.**JobParseGz**(*break_duration*)

Bases: object

Stores the persistent information over several jobs and exposes a job method that only takes the varying parts as one argument.

Note: Using a closure is not possible as it is not pickable / serializable.

Methods

run	

run(*filename*)

pyndl.corpus.**create_corpus_from_gz**(*directory*, *outfile*, ***, *n_threads=1*, *verbose=False*)

Create a corpus file from a set of gunziped (.gz) files in a directory.

Parameters

directory
[str] use all gz-files in this directory and all subdirectories as input.

outfile
[str] name of the outfile that will be created.

n_threads
[int] number of threads to use.

verbose
[bool]

pyndl.corpus.**read_clean_gzfile**(*gz_file_path*, ***, *break_duration=2.0*)

Generator that opens and reads a gunzipped xml subtitle file, while all xml tags and timestamps are removed.

Parameters

break_duration
[float] defines the amount of time in seconds that need to pass between two subtitles in order to start a new paragraph in the resulting corpus.

Yields

line
[non empty, cleaned line out of the xml subtitle file]

Raises

FileNotFoundError
[if file is not there.]

pyndl.correlation.**correlation**(*semantics*, *activations*, *, *verbose=False*, *allow_nan=False*)

calculates the correlations between the semantics and the activations.

Returns

np.array (n_outcomes, n_events)
The first column contains all correlations between the first event and all possible outcomes in the semantcs.
The first column reads like:

0. **correlation between first event and first outcome in the semantic** (gold standard) space.
1. **correlation between first event and second outcome …**

…

pyndl.count

pyndl.count provides functions in order to count

- words and symbols in a corpus file
- cues and outcomes in an event file

class pyndl.count.**CuesOutcomes**(*n_events*, *cues*, *outcomes*)

Bases: tuple

Attributes

cues
Alias for field number 1

n_events
Alias for field number 0

outcomes
Alias for field number 2

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

cues

Alias for field number 1

n_events

Alias for field number 0

outcomes

Alias for field number 2

class pyndl.count.**WordsSymbols**(*words*, *symbols*)

Bases: tuple

Attributes

symbols
Alias for field number 1

words
Alias for field number 0

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

symbols

Alias for field number 1

words

Alias for field number 0

pyndl.count.**cues_outcomes**(*event_file_name*, ***, *n_jobs=2*, *number_of_processes=None*, *verbose=False*)

Counts cues and outcomes in event_file_name using n_jobs processes.

Returns

(n_events, cues, outcomes)
[(int, collections.Counter, collections.Counter)]

pyndl.count.**load_counter**(*filename*)

Loads a counter out of a tab delimitered text file.

pyndl.count.**save_counter**(*counter*, *filename*, ***, *header='key\tfreq\n'*)

Saves a counter object into a tab delimitered text file.

pyndl.count.**words_symbols**(*corpus_file_name*, ***, *n_jobs=2*, *number_of_processes=None*, *lower_case=False*, *verbose=False*)

Counts words and symbols in corpus_file_name using n_jobs processes.

Returns

(words, symbols)
[(collections.Counter, collections.Counter)]

pyndl.io

pyndl.io provides functions to create event generators from different sources in order to use them with *pyndl.ndl* to train NDL models or to save existing events from a DataFrame or a list to a file.

pyndl.io.**events_from_dataframe**(*df*, *columns=('cues', 'outcomes')*)

Yields events for all events in a pandas dataframe.

Parameters

df
[pandas.DataFrame] a pandas DataFrame with one event per row and one colum with the cues and one column with the outcomes.

columns
[tuple] a tuple of column names

Yields

cues, outcomes
[list, list] a tuple of two lists containing cues and outcomes

pyndl.io.**events_from_file**(*event_path*, *compression='gzip'*, *start=0*, *step=1*)

Yields events for all events in a gzipped event file.

Parameters

event_path
[str] path to gzipped event file

compression
[str] indicates whether the events should be read from gunzip file or not can be {"gzip" or None}

start: int
first event to read

step: int
slice every step-th event (useful for parallel computations)

Yields

cues, outcomes
[list, list] a tuple of two lists containing cues and outcomes

pyndl.io.**events_from_list**(*lst*)

Yields events for all events in a list.

Parameters

lst
[list of list of str or list of str] a list either containing a list of cues as strings and a list of outcomes as strings or a list containing a cue and an outcome string, where cues respectively outcomes are seperated by an undescore

Yields

cues, outcomes
[list, list] a tuple of two lists containing cues and outcomes

pyndl.io.**events_to_file**(*events*, *file_path*, *delimiter='\t'*, *compression='gzip'*, *columns=('cues', 'outcomes')*, *compatible=False*)

Writes events to a file

Parameters

events
[pandas.DataFrame or Iterator or Iterable] a pandas DataFrame with one event per row and one colum with the cues and one column with the outcomes or a list of cues and outcomes as strings or a list of a list of cues and a list of outcomes which should be written to a file

file_path: str
path to where the file should be saved

delimiter: str
Seperator which should be used. Default ist a tab

compression
[str] indicates whether the events should be read from gunzip file or not can be {"gzip" or None}

columns: tuple
a tuple of column names

compatible: bool
if true add a third frequency column (all ones) for compatibility with ndl2

pyndl.io.**safe_write_path**(*path*, *template='{path.stem}-{counter}{path.suffix}'*)

Create a file path to avoid overwriting existing files. Returns the original path if it does not exist or an incremented version according to the template.

This function with the default template creates filenames like pathname/example.png, pathname/example-1.png, pathname/example-2.png, ...

Parameters

path: file path
template: format string syntax of incremented file name.
available variables are counter (int) and path (pathlib.Path).

Returns

path: the input path or (if file exists) the path with incremented filename.

pyndl.ndl

pyndl.ndl provides functions in order to train NDL models

class pyndl.ndl.**WeightDict**(**args*, ***kwargs*)

Bases: defaultdict

Subclass of defaultdict to represent outcome-cue weights.

Notes

Weight for each outcome-cue combination is 0 per default.

Attributes

attrs
default_factory
Factory for default value called by __missing__().

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(key[, default])	If key is not found, default is returned if given, otherwise KeyError is raised
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

property attrs

pyndl.ndl.**data_array**(*weights*, *, *attrs=None*)

Calculate the weights for all_outcomes over all events in event_file.

Parameters

weights
[dict of dicts of floats or WeightDict] the first dict has outcomes as keys and dicts as values the second dict has cues as keys and weights as values weights[outcome][cue] gives the weight between outcome and cue. If a dict of dicts is given, attrs is required. If a WeightDict is given, attrs is optional

attrs
[dict] A dictionary of attributes

Returns

weights
[xarray.DataArray] with dimensions 'outcomes' and 'cues'. You can lookup the weights between a cue and an outcome with `weights.loc[{'outcomes': outcome, 'cues': cue}]` or `weights.loc[outcome].loc[cue]`.

pyndl.ndl.**dict_ndl**(*events*, *alphas*, *betas*, *lambda_=1.0*, *, *weights=None*, *inplace=False*, *remove_duplicates=None*, *make_data_array=False*, *verbose=False*)

Calculate the weights for all_outcomes over all events in event_file.

This is a pure python implementation using dicts.

Parameters

events
[generator or str] generates cues, outcomes pairs or the path to the event file

alphas
[dict or float] a (default)dict having cues as keys and a value below 1 as value

betas
[(float, float)] one value for successful prediction (reward) one for punishment

lambda_
[float]

weights
[dict of dicts or xarray.DataArray or None] initial weights

inplace: {True, False}
if True calculates the weightmatrix inplace if False creates a new weightmatrix to learn on

remove_duplicates
[{None, True, False}] if None though a ValueError when the same cue is present multiple times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

make_data_array
[{False, True}] if True makes a xarray.DataArray out of the dict of dicts.

verbose
[bool] print some output if True.

Returns

weights
[dict of dicts of floats] the first dict has outcomes as keys and dicts as values the second dict has cues as keys and weights as values weights[outcome][cue] gives the weight between outcome and cue.

or
weights
[xarray.DataArray] with dimensions 'outcomes' and 'cues'. You can lookup the weights between a cue and an outcome with `weights.loc[{'outcomes': outcome, 'cues': cue}]` or `weights.loc[outcome].loc[cue]`.

Notes

The metadata will only be stored when *make_data_array* is True and then *dict_ndl* cannot be used to continue learning. At the moment there is no proper way to automatically store the meta data into the default dict.

`pyndl.ndl.ndl`(*events*, *alpha*, *betas*, *lambda_=1.0*, ***, *method='openmp'*, *weights=None*, *number_of_threads=None*, *n_jobs=8*, *len_sublists=None*, *n_outcomes_per_job=10*, *remove_duplicates=None*, *verbose=False*, *temporary_directory=None*, *events_per_temporary_file=10000000*)

Calculate the weights for all_outcomes over all events in event_file given by the files path.

This is a parallel python implementation using numpy, multithreading and the binary format defined in preprocess.py.

Parameters

events
[generator or str] generates cues, outcomes pairs or the path to the event file

alpha
[float] saliency of all cues

betas
[(float, float)] one value for successful prediction (reward) one for punishment

lambda_
[float]

method
[{'openmp', 'threading'}]

weights
[None or xarray.DataArray] the xarray.DataArray needs to have the named dimensions 'cues' and 'outcomes'

n_jobs
[int] a integer giving the number of threads in which the job should executed

n_outcomes_per_job
[int] a integer giving the length of sublists generated from all outcomes

remove_duplicates
[{None, True, False}] if None though a ValueError when the same cue is present multiple times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

verbose
[bool] print some output if True.

temporary_directory
[str] path to directory to use for storing temporary files created; if none is provided, the operating system's default will be used (/tmp on unix)

events_per_temporary_file: int
Number of events in each temporary binary file. Has to be larger than 1

Returns

weights
[xarray.DataArray] with dimensions 'outcomes' and 'cues'. You can lookup the weights between a cue and an outcome with `weights.loc[{'outcomes': outcome, 'cues': cue}]` or `weights.loc[outcome].loc[cue]`.

pyndl.ndl.**slice_list**(*list_*, *len_sublists*)

Slices a list in sublists with the length len_sublists.

Parameters

list_
[list] list which should be sliced in sublists

len_sublists
[int] integer which determines the length of the sublists

Returns

seq_list
[list of lists] a list of sublists with the length len_sublists

pyndl.preprocess

pyndl.preprocess provides functions in order to preprocess data and create event files from it.

class pyndl.preprocess.**JobFilter**(*keep_cues*, *keep_outcomes*, *remove_cues*, *remove_outcomes*, *cue_map*, *outcome_map*)

Bases: `object`

Stores the persistent information over several jobs and exposes a job method that only takes the varying parts as one argument.

Note: Using a closure is not possible as it is not pickable / serializable.

Methods

job	
process_cues	
process_cues_all	
process_cues_keep	
process_cues_map	
process_cues_remove	
process_outcomes	
process_outcomes_all	
process_outcomes_keep	
process_outcomes_map	
process_outcomes_remove	
return_empty_string	

job(*line*)

process_cues(*cues*)

process_cues_all(*cues*)

process_cues_keep(*cues*)

process_cues_map(*cues*)

process_cues_remove(*cues*)

process_outcomes(*outcomes*)

process_outcomes_all(*outcomes*)

process_outcomes_keep(*outcomes*)

process_outcomes_map(*outcomes*)

process_outcomes_remove(*outcomes*)

static return_empty_string()

pyndl.preprocess.**bandsample**(*population*, *sample_size=50000*, ***, *cutoff=5*, *seed=None*, *verbose=False*)

Creates a sample of size sample_size out of the population using band sampling.

pyndl.preprocess.**create_binary_event_files**(*event_file*, *path_name*, *cue_id_map*, *outcome_id_map*, ***, *sort_within_event=False*, *n_jobs=2*, *events_per_file=10000000*, *overwrite=False*, *remove_duplicates=None*, *verbose=False*)

Creates the binary event files for a tabular cue outcome corpus.

Parameters

event_file
[str] path to tab separated text file that contains all events in a cue outcome table.

path_name
[str] folder name where to store the binary event files

cue_id_map
[dict (str -> int)] cue to id map

outcome_id_map
[dict (str -> int)] outcome to id map

sort_within_event
[bool] should we sort the cues and outcomes within the event

n_jobs
[int] number of threads to use

events_per_file
[int] Number of events in each binary file. Has to be larger than 1

overwrite
[bool] overwrite files if they exist

remove_duplicates
[{None, True, False}] if None though a ValueError when the same cue is present multiple times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

verbose
[bool]

Returns

number_events
[int] sum of number of events written to binary files

pyndl.preprocess.**create_event_file**(*corpus_file*, *event_file*, ***, *allowed_symbols='*'*, *context_structure='document'*, *event_structure='consecutive_words'*, *event_options=(3,)*, *cue_structure='trigrams_to_word'*, *lower_case=False*, *remove_duplicates=True*, *verbose=False*)

Create an text based event file from a corpus file.

Warning: '_', '#', and ' ' are removed from the input of the corpus file and replaced by a ' ', which is treated as a word boundary.

Parameters

corpus_file
[str] path where the corpus file is

event_file
[str] path where the output file will be created

allowed_symbols
[str, function] all allowed symbols to include in the events as a set of characters. The set of characters might be explicit or contains Regex character sets.

'_', '#', and TAB are special symbols in the event file and will be removed automatically. If the corpus file contains these special symbols a warning will be given.

These examples define the same allowed symbols:

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
'a-zA-Z'
'*'
```

or a function indicating which characters to include. The function should return *True*, if the passed character is a allowed symbol.

For example:

```
lambda chr: chr in
↪"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
lambda chr: ('a' <= chr <= 'z') or ('A' <= chr <= 'Z')
```

context_structure
[{"document", "paragraph", "line"}]

event_structure
[{"line", "consecutive_words", "word_to_word", "sentence"}]

event_options
[None or (number_of_words,) or (before, after) or None] in "consecutive words" the number of words of the sliding window as an integer; in "word_to_word" the number of words before and after the word of interest each as an integer.

cue_structure: {"trigrams_to_word", "word_to_word", "bigrams_to_word"}
lower_case
[bool] should the cues and outcomes be lower cased

remove_duplicates
[bool] create unique cues and outcomes per event

verbose
[bool]

Notes

Breaks / Separators :

What marks parts, where we do not want to continue learning?

- `---end.of.document---` string?
- line breaks?
- empty lines?

What do we consider one event?

- three consecutive words?
- one line of the corpus?
- everything between two empty lines?
- everything within one document?

Should the events be connected to the events before and after?

No.

Context :

A context is a whole document or a paragraph within which we will take (three) consecutive words as occurrences or events. The last words of a context will not form an occurrence with the first words of the next context.

Occurrence :

An occurrence or event is will result in one event in the end. This can be (three) consecutive words, a sentence, or a line in the corpus file.

`pyndl.preprocess.`**`event_generator`**(*event_file*, *cue_id_map*, *outcome_id_map*, *, *sort_within_event=False*)

`pyndl.preprocess.`**`filter_event_file`**(*input_event_file*, *output_event_file*, *, *keep_cues='all'*, *keep_outcomes='all'*, *remove_cues=None*, *remove_outcomes=None*, *cue_map=None*, *outcome_map=None*, *n_jobs=1*, *number_of_processes=None*, *chunksize=100000*, *verbose=False*)

Filter an event file by a list or a map of cues and outcomes.

Parameters

You can either use keep_*, remove_*, or map_*.

input_event_file
[str] path where the input event file is

output_event_file
[str] path where the output file will be created

keep_cues
["all" or sequence of str] list of all cues that should be kept

keep_outcomes
["all" or sequence of str] list of all outcomes that should be kept

remove_cues
[None or sequence of str] list of all cues that should be removed

remove_outcomes
[None or sequence of str] list of all outcomes that should be removed

cue_map
[dict] maps every cue as key to the value. Removes all cues that do not have a key. This can be used to map several different cues to the same cue or to rename cues.

outcome_map
[dict] maps every outcome as key to the value. Removes all outcome that do not have a key. This can be used to map several different outcomes to the same outcome or to rename outcomes.

n_jobs
[int] number of threads to use

chunksize
[int] number of chunks per submitted job, should be around 100000

Notes

It will keep all cues that are within the event and that (for a human reader) might clearly belong to a removed outcome. This is on purpose and is the expected behaviour as these cues are in the context of this outcome.

If an event has no cues it gets removed, but if an event has no outcomes it is still present in order to capture the background rate of that cues.

pyndl.preprocess.**ngrams_to_word**(*occurrences*, *n_chars*, *outfile*, *remove_duplicates=True*)

Process the occurrences and write them to outfile.

Parameters

occurrences
[sequence of (cues, outcomes) tuples] cues and outcomes are both strings where underscores and # are special symbols.

n_chars
[number of characters (e.g. 2 for bigrams, 3 for trigrams, ...)]

outfile
[file handle]

remove_duplicates
[bool] if True make cues and outcomes per event unique

pyndl.preprocess.**process_occurrences**(*occurrences*, *outfile*, ***, *cue_structure='trigrams_to_word'*, *remove_duplicates=True*)

Process the occurrences and write them to outfile.

Parameters

occurrences
[sequence of (cues, outcomes) tuples] cues and outcomes are both strings where underscores and # are special symbols.

outfile
[file handle]

cue_structure
[{'bigrams_to_word', 'trigrams_to_word', 'word_to_word'}]

remove_duplicates
[bool] if True make cues and outcomes per event unique

pyndl.preprocess.**read_binary_file**(*binary_file_path*)

pyndl.preprocess.**to_bytes**(*int_*)

pyndl.preprocess.**to_integer**(*byte_*)

pyndl.preprocess.**write_events**(*events*, *filename*, ***, *start=0*, *stop=4294967295*, *remove_duplicates=None*)

Write out a list of events to a disk file in binary format.

Parameters

events
[iterator of (cue_ids, outcome_ids) tuples called event]

filename
[string]

start
[first event to write (zero based index)]

stop
[last event to write (zero based index; excluded)]

remove_duplicates
[{None, True, False}] if None though a ValueError when the same cue is present multiple times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

Returns

number_events
[int] actual number of events written to file

Raises

StopIteration
[events generator is exhausted before stop is reached]

Notes

The **binary format** as the following structure:

```
8 byte header
nr of events
nr of cues in first event
ids for every cue
nr of outcomes in first event
ids for every outcome
nr of cues in second event
...
```

pyndl.wh

pyndl.wh provides functions in order to train Widrow-Hoff (WH) models. In contrast to the Rescorla-Wagner (RW) models, the WH models can not only have binary cues and outcomes, but can encode gradual intensities in the cues and outcomes. This is done by associating a vector of continues values (real numbers) to each cue and outcome. The size of the vector has to be the same for all cues and for all outcomes, but can differ between cues and outcomes.

It is possible to calculate weights for continuous cues or continues outcomes, while keeping the outcomes respectively cues binary. Finally, it is possible to have both sides, cues and outcomes, to be continues and calculate the Widrow-Hoff learning rule between them.

pyndl.wh.**dict_wh**(*events*, *eta*, *cue_vectors*, *outcome_vectors*, *, *weights=None*, *inplace=False*, *remove_duplicates=None*, *make_data_array=False*, *verbose=False*)

Calculate the weights for all_outcomes over all events in events.

This is a pure python implementation using dicts.

Parameters

events
[generator or str] generates cues, outcomes pairs or the path to the event file

eta
[float] learning rate

cue_vectors
[xarray.DataArray] matrix that contains the cue vectors for each cue

outcome_vectors
[xarray.DataArray] matrix that contains the target vectors for each outcome

weights
[dict of dicts or xarray.DataArray or None] initial weights

inplace: {True, False}
if True calculates the weightmatrix inplace if False creates a new weightmatrix to learn on

remove_duplicates
[{None, True, False}] if None though a ValueError when the same cue is present multiple times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

make_data_array
[{False, True}] if True makes a xarray.DataArray out of the dict of dicts.

verbose
[bool] print some output if True.

Returns

weights
[dict of dicts of floats] the first dict has outcomes as keys and dicts as values the second dict has cues as keys and weights as values weights[outcome][cue] gives the weight between outcome and cue.

or
weights
[xarray.DataArray] with dimensions 'outcome_vector_dimensions' and 'cue_vector_dimensions'. You can lookup the weights between a cue dimension and an outcome dimension with `weights.loc[{'outcome_vector_dimensions': outcome_vector_dimension,`

'cue_vector_dimensions': cue_vector_dimension}] or weights.loc[outcome_vector_dimension].loc[cue_vector_dimension].

Notes

The metadata will only be stored when *make_data_array* is True and then *dict_ndl* cannot be used to continue learning. At the moment there is no proper way to automatically store the meta data into the default dict.

Furthermore, this implementation only supports the 'real to real' case where cue vectors are learned on outcome vectors. For the 'binary to real' or 'real to binary' cases the *wh.wh* function needs to be used which uses a fast cython implementation.

The main purpose of this function is to have a reference implementation which is used to validate the faster cython version against. Additionally, this function can be a good starting point to develop different flavors of the Widrow-Hoff learning rule.

pyndl.wh.**wh**(*events*, *eta*, *, *cue_vectors=None*, *outcome_vectors=None*, *method='openmp'*, *weights=None*, *n_jobs=8*, *n_outcomes_per_job=10*, *remove_duplicates=None*, *verbose=False*, *temporary_directory=None*, *events_per_temporary_file=10000000*)

Calculate the weights for all events using the Widrow-Hoff learning rule in three different flavors.

In the first flavor, cues and outcomes both are vectors and the names in the eventfiles refer to these vectors. The vectors for all cues and outcomes are given as an xarray.DataArray with the arguments *cue_vectors* and `outcome_vectors'.

In the second and third flavor, only the cues or only the outcomes are treated as vectors and the ones not being treated as vectors are still considered being present or not being present in a binary way.

This is a parallel python implementation using cython, numpy, multithreading and the binary format defined in preprocess.py.

Parameters

events
[str] path to the event file

eta
[float] learning rate

cue_vectors
[xarray.DataArray] matrix that contains the cue vectors for each cue

outcome_vectors
[xarray.DataArray] matrix that contains the target vectors for each outcome

method
[{'openmp', 'threading', 'numpy'}] 'numpy' works only for real to real Widrow-Hoff.

weights
[None or xarray.DataArray] the xarray.DataArray needs to have the named dimensions 'cues' or 'cue_vector_dimensions' and 'outcomes' or 'outcome_vector_dimensions'

n_jobs
[int] an integer giving the number of threads in which the job should be executed

n_outcomes_per_job
[int] an integer giving the number of outcomes that are processed in one job

remove_duplicates
[{None, True, False}] if None raise a ValueError when the same cue is present multiple

times in the same event; True make cues and outcomes unique per event; False keep multiple instances of the same cue or outcome (this is usually not preferred!)

verbose
[bool] print some output if True

temporary_directory
[str] path to directory to use for storing temporary files created; if none is provided, the operating system's default will be used like '/tmp' on unix

events_per_temporary_file: int
Number of events in each temporary binary file. Has to be larger than 1

Returns

weights
[xarray.DataArray] the dimensions of the weights reflect the type of Widrow-Hoff that was run (real to real, binary to real, real to binary or binary to binary). The dimension names reflect this in the weights. They are a combination of 'outcomes' x 'outcome_vector_dimensions' and 'cues' x 'cue_vector_dimensions' with dimensions 'outcome_vector dimensions' and 'cue_vector_dimensions'. You can lookup the weights between a vector dimension and a cue with `weights.loc[{'outcome_vector_dimensions': outcome_vector_dimension, 'cue_vector_dimensions': cue_vector_dimension}]` or `weights.loc[vector_dimension].loc[cue_vector_dimension]`.

Development

Getting Involved

The *pyndl* project welcomes help in the following ways:

- Making Pull Requests for code, tests or documentation.
- Commenting on open issues and pull requests.
- Helping to answer questions in the issue section.
- Creating feature requests or adding bug reports in the issue section.

Prerequisites

To make changes to the *pyndl* code base the following prerequisites need to be fulfilled on your machine:

- you have Python3 installed
- you have Cython installed and can compile Cython extensions (*conda install cython* should do the trick, but sometimes this can be a little bit tricky)
- you have poetry installed
- you have *git* installed

Note: Depending on your operating system and your architecture properly installing Cython and being able to compile Cython extensions can be a bit tricky. If the installation of python fails, it is a good first step to check that the Cython installation is done properly via *conda* or through you package manager.

Workflow

1. Fork this repository on Github. From here on we assume you successfully forked this repository to https://github.com/yourname/pyndl.git
2. Install all dependencies with poetry (https://python-poetry.org/)

```
git clone https://github.com/yourname/pyndl.git
cd pyndl
poetry install
```

3. Add code, tests or documentation.
4. Test your changes locally by running within the root folder (`pyndl/`)

```
poetry run pytest
poetry run pylint pyndl
```

5. Add and commit your changes after tests run through without complaints.

```
git add -u
git commit -m 'fixes #42 by posing the question in the right way'
```

   You can reference relevant issues in commit messages (like #42) to make GitHub link issues and commits together, and with phrase like "fixes #42" you can even close relevant issues automatically.

6. Push your local changes to your fork:

```
git push git@github.com:yourname/pyndl.git
```

7. Open the Pull Requests page at https://github.com/yourname/pyndl/pulls and click "New pull request" to submit your Pull Request to https://github.com/quantling/pyndl.

Running tests

We use `poetry` to manage testing. You can run the tests by executing the following within the repository's root folder (`pyndl/`):

```
poetry run pytest
```

For extensive, time and memory consuming tests run (at least 12 GB of free memory should be available):

```
poetry run pytest --run-slow
```

For manually checking coding guidelines run:

```
poetry run pylint pyndl
```

The linting gives still a lot of complaints that need some decisions on how to fix them appropriately.

Note: Previous versions of *pyndl* used `make` and `tox` to manage testing. For documentation on this, please check the respective version documentations

---

Local testing with conda

Sometimes it might be useful to test if `pyndl` works in a clean python environment. Besides `poetry` this is possible with `conda` as well. The commands are as follows:

```
conda create -n testpyndl
conda activate testpyndl
conda install python
python -c 'from pyndl import ndl; print("success")'  # this should fail
git clone https://github.com/quantling/pyndl.git
pip install pyndl
python -c 'from pyndl import ndl; print("success")'  # this should succeed
conda deactivate
conda env remove -n testpyndl
```

Memory profiling

Sometimes it is useful to monitory the memory footprint of the python process. This can be achieved by using `memory_profiler` (https://pypi.python.org/pypi/memory_profiler).

CPU profiling of C extensions

In order to profile Cython or C extensions that are invoked from python `yep` is a good tool to do that. `yep` builds ontop of `google-perftools`. (https://pypi.org/project/yep/)

Keeping a fork in sync with main

Note: If you have questions regarding `git` it is mostly a good start to read up on it on github help pages, i. e. https://help.github.com/articles/working-with-forks/ .

If you fork the `pyndl` project on github.com you might want to keep it in sync with main. In order to do so, you need to setup a remote repository within a local `pyndl` clone of you fork. This remote repository will point to the original `pyndl` repository and is usually called `upstream`. In order to do so run with a Terminal within the cloned pyndl folder:

```
git remote add upstream https://github.com/quantling/pyndl.git
```

After having set up the `upstream` repository you can manually sync your local repository by running:

```
git fetch upstream
```

In order to sync you `main` branch run:

```
git checkout main
git merge upstream/main
```

If the merge cannot be fast-forward, you should resolve any issue now and commit the manually merged files.

After that you should sync you local repository with you github fork by running:

```
git push
```

Some sources with more explanation:

- https://help.github.com/articles/configuring-a-remote-for-a-fork/
- https://help.github.com/articles/syncing-a-fork/

Building documentation

Building the documentation requires some extra dependencies. Usually, these are installed when installing the dependencies with poetry. Some services like Readthedocs, however, require the documentation dependencies extra. For that reason, they can also be found in *docs/requirements.txt*. For normal usage, installing all dependencies with poetry is sufficient.

The projects documentation is stored in the `pyndl/docs/` folder and is created with `sphinx`. However, it is not necessary to build the documentation from there.

You can rebuild the documentation by either executing

```
poetry run sphinx-build -b html docs/source docs/build/html
```

in the repository's root folder (`pyndl`) or by executing

```
poetry run make html
```

in the documentation folder (`pyndl/docs/`).

Continuous Integration

We use several services in order to continuously monitor our project:

Service	Status	Config file	Description
Github Actions		python-test.yml	Automated testing
Codecov			Monitoring of test coverage
LGTM			Monitoring code quality

Licensing

All contributions to this project are licensed under the MIT license. Exceptions are explicitly marked. All contributions will be made available under MIT license if no explicit request for another license is made and agreed on.

Release Process

1. Update the version accordingly to *Versioning* below. This can be easily done by poetry running

   ```
   poetry version major|minor|patch|...
   ```

2. Merge Pull Requests with new features or bugfixes into *pyndl*'s' `main` branch.
3. Create a new release on Github of the *main* branch of the form `vX.Y.Z` (where `X`, `Y`, and `Z` refer to the new version). Add a description of the new feature or bugfix. For details on the version number see *Versioning* below. This will trigger a Action to automatically build and upload the release to PyPI
4. Check if the new version is on pypi (https://pypi.python.org/pypi/pyndl/).

Versioning

We use a semvers versioning scheme. Assuming the current version is `X.Y.Z` than `X` refers to the major version, `Y` refers to the minor version and `Z` refers to a bugfix version.

Bugfix release

For a bugfix only merge, which does not add any new features and does not break any existing API increase the bugfix version by one (`X.Y.Z -> X.Y.Z+1`).

Minor release

If a merge adds new features or breaks with the existing API a deprecation warning has to be supplied which should keep the existing API. The minor version is increased by one (`X.Y.Z -> X.Y+1.Z`). Deprecation warnings should be kept until the next major version. They should warn the user that the old API is only usable in this major version and will not be available any more with the next major `X+1.0.0` release onwards. The deprecation warning should give the exact version number when the API becomes unavailable and the way of achieving the same behaviour.

Major release

If enough changes are accumulated to justify a new major release, create a new pull request which only contains the following two changes:

- the change of the version number from `X.Y.Z` to `X+1.0.0`
- remove all the API with deprecation warning introduced in the current `X.Y.Z` release

Credits

Authors

pyndl was mainly developed by Konstantin Sering, Marc Weitz, David-Elias Künstle, Lennart Schneider and Elnaz Shafaei-Bajestan. For the full list of contributors have a look at Github's Contributor summary.

Currently, it is maintained by Konstantin Sering and Marc Weitz.

Contact

In case you want to contact the project maintainers, please send an email to

> konstantin [dot] sering [at] uni [minus] tuebingen [dot de

Citation

If this work was helpful in your work, feel free to cite it as

> Konstantin Sering, Marc Weitz, David-Elias Künstle, Lennart Schneider, & Elnaz Shafaei-Bajestan. (2022). Pyndl: Naive discriminative learning in python. http://doi.org/10.5281/zenodo.597964

If you are using BibTex you may want to use this example BibTex entry:

```
@misc{pyndl,
      author       = {Konstantin Sering and
                      Marc Weitz and
                      David-Elias Künstle and
                      Lennart Schneider and
                      Elnaz Shafaei-Bajestan},
      title        = {Pyndl: Naive discriminative learning in python},
      year         = 2017,
      doi          = {10.5281/zenodo.597964},
      url          = {https://doi.org/10.5281/zenodo.597964}
}
```

Note: If you want to cite a specific version, check out the history on zenodo!

Funding

pyndl has been supported by the Alexander von Humboldt Professorship awarded to R. Harald Baayen, the ERC advanced Grant WIDE (no. 742545), and the University of Tübingen.

Acknowledgements

This package is build as a Python replacement for the R ndl2 package. Some ideas on how to build the API and how to efficiently run the Rescorla Wagner iterative learning on large text corpora are inspired by the ndl2 package.

PYTHON MODULE INDEX

p

INDEX

R

S

T

W