TraceableDict Documentation

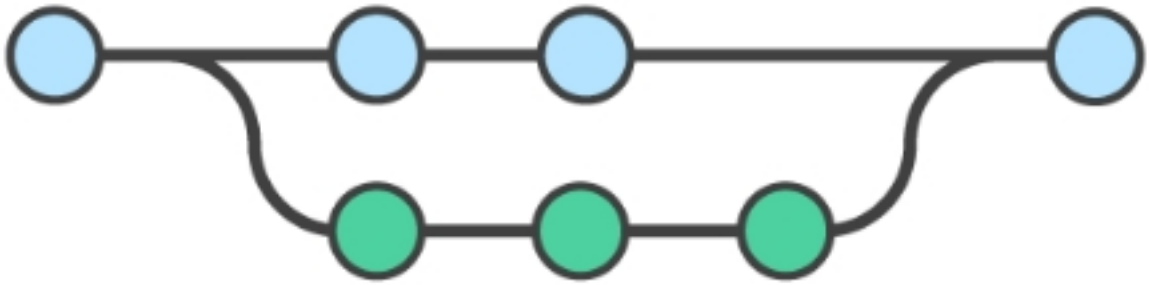
Release 1.0

Shahar Azulay, Rinat Ishak

Oct 16, 2018

Overview

Traceable Python dictionary, that stores change history in an efficient way inside the object.



Shahar Azulay, Rinat Ishak

Usage Examples:

Create a traceable dictionary

```
>>> from traceable_dict import TraceableDict
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>> D1.revisions
[]
```

Commit the dictionary for the first time

```
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': [1]}
>>> D1.revisions
[1]
>>> D1.has_uncommitted_changes
False
```

Update the dictionary while tracing the changes

```
>>> D1['new_key'] = 'new_value'
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
True
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
```

(continues on next page)

(continued from previous page)

```
False
>>> D1.revisions
[1, 2]
```

Checkout previous revisions

```
>>> D1.as_dict()
{'old_key': 'old_value', 'new_key': 'new_value'}
>>>
>>> D_original = D1.checkout(revision=1)
>>> D_original.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 1

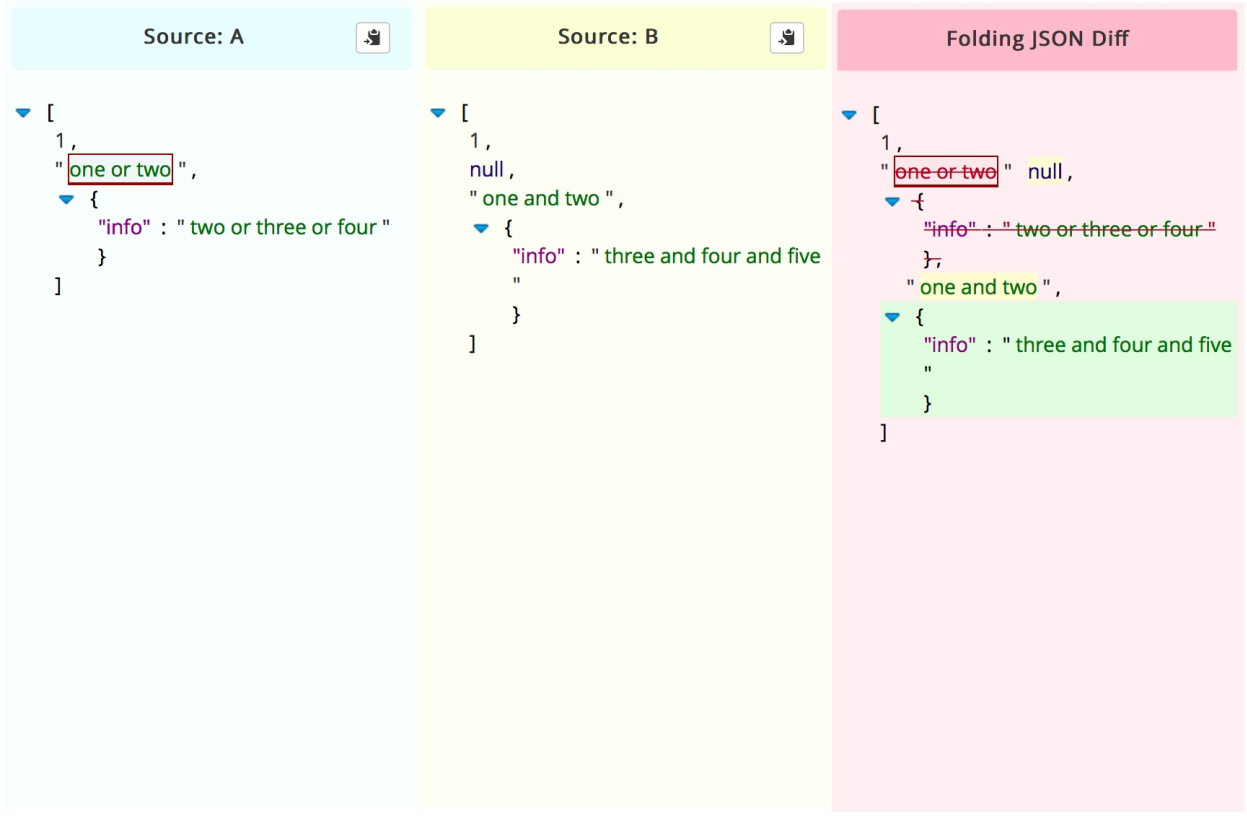
General Concept

Often a Python dictionary object is used to represent a pre-known structured data that captures some state of a system. A non-relational database such as MongoDB is a great example of such use-case, where the BSON-based documents can easily be loaded into a Python dictionary. Those dict-like documents help store complex information, whose structure may change over time, and are highly common in the industry.

In cases where the dictionary or JSON-like structure represents a meaningful state of a system, tracing it changes may be a highly valueble part in the monitoring of the system.

This module implements a traceable Python dictionary, that stores change history in an efficient way inside the object. It allows the user to:

1. **Trace reverions** of the dictionary's content and structure.
2. **Roll the dictionary back** to previously stored values.
3. **Trace the changes** between its different revisions.
4. **Revert** unwanted changes made.
5. **Provide a meaningful id** to the revisions - such as a timestamp, or verison number.
6. More....



[1] tracing the changes in a JSON-like object

CHAPTER 2

The Solution

The major problem in creating a proper Stacking ensemble is getting it right. The wrong way to perform stacking would be to

1. **Train** the first level models over the target.
2. Get the first level models predictions over the inputs.
3. **Train** the meta-level Stacker over the predictions of the first level models.

Why would that be the wrong way to go?

Overfitting

Our meta-level regressor would be exposed to severe overfitting from one of the first level models. For example, if one of five first level models would be highly overfitted to the target, practically "storing" the y target it is showns in train time for test time. The meta-level model, trained over the same target would see this model as excellent - predicting the target y with impressive accuracy almost everytime.

This will result in a hight weight to this model, making the entire pipeline useless in test time.

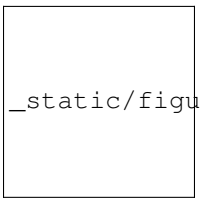
CHAPTER 3

Performance

The solution is never using the train abilities of the first level model - but using their abilities in test. What does it mean? it means the meta-level model would never be exposed to a y_hat generated by any first level model where the actual target sample representing this y_hat in the data was given to that model in training.

Each model will deliever its predictions in a “cross_val_predict” manner (in sklearn terms). If it’s a great model, it will demonstrate great generalization skills making its test-time predictions valuable to the meta-level regressor. If it’s a highly overfitted model - the test-time predictions it will hand down the line will be showns for their true abilities, causing it to recieve a low weight.

How do we achieve that? internal cross validation.



[1] achienving stacking ensemble using internal cross-validation

CHAPTER 4

Creating the traceable dict for the first time

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'first_key': 'first_value'}
>>>
>>> D1 = TraceableDict(d1)
>>> D1
{'first_key': 'first_value', '__trace__': {}, '__revisions__': []}
>>> D1.revisions
[]
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>>
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
>>>
[1]
```

CHAPTER 5

Updating a single key inside the dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>>
>>> D1.has_uncommitted_changes
True
>>>
>>> D1.commit(revision=1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': [1]}
>>> D1.revisions
[1]
>>> D1.has_uncommitted_changes
False
>>> D1['new_key'] = 'new_val'
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
True
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'new_key'), None, '__a__')]}
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
[1, 2]
```

CHAPTER 6

Updating the entire dictionary while tracing the changes

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1
{'old_key': 'old_value', '__trace__': {}, '__revisions__': []}
>>>
>>> D1.commit(revision=1)
>>> D1.trace
{}
>>> d2 = {'old_key': 'updated_value', 'new_key': 'new_value'}
>>>
>>> D1 = D1 | d2
>>> D1.as_dict()
{'new_key': 'new_value', 'old_key': 'updated_value'}
>>> D1.trace
{'_uncommitted_': [(('_root_', 'old_key'), 'old_value', '__u__'), (('_root_', 'new_key
↪'), None, '__a__')]}
>>>
>>> D1.commit(revision=2)
>>> D1.trace
{'2': [(('_root_', 'old_key'), 'old_value', '__u__'), (('_root_', 'new_key'), None, '_
↪_a__')]}
>>> D1.has_uncommitted_changes
False
>>> D1.revisions
[1, 2]
```

CHAPTER 7

Reverting un-committed changes to a dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1['new_key'] = 'new_val'
>>> D1.as_dict()
{'new_key': 'new_val', 'old_key': 'old_value'}
>>> D1.trace
{'_uncommitted_': [(('_root_', 'new_key'), None, '__a__')]}
>>>
>>> D1.revert()
>>>
>>> D1.has_uncommitted_changes
False
>>> D1.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 8

Checkout previous revisions of the dictionary

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> d2 = {'old_key': 'updated_value', 'new_key': 'new_value'}
>>>
>>> D1 = D1 | d2
>>> D1.as_dict()
{'new_key': 'new_value', 'old_key': 'updated_value'}
>>>
>>> D1.commit(revision=2)
>>> D1.revisions
[1, 2]
>>>
>>> D_original = D1.checkout(revision=1)
>>> D_original.as_dict()
{'old_key': 'old_value'}
```

CHAPTER 9

Displaying the commit logs over the different revisions

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'key1': 'value1'}
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1['key1'] = 'new_value1'
>>> D1.commit(revision=2)
>>>
>>> log = D1.log(path=('key1',))
changeset:   1
value:       {'key1': 'value1'}


changeset:   2
value:       {'key1': 'new_value1'}


>>> log
{1: {'key1': 'value1'}, 2: {'key1': 'new_value1'}}
```

CHAPTER 10

Show changes between revisions, or latest revision and working tree

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {
... 'key1': 'value1',
... 'key2': 'value2'
>>> }
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> d2 = {
... 'key1': 'new_value1',
... 'key3': 'value3'
>>> }
>>>
>>> D1 = D1 | d2
>>> D1.commit(revision=2)
>>>
>>> diff = D1.diff(revision=2)
>>> diff
{'key1': '---------value1 +++++++++new_value1',
 'key2': '---------value2',
 'key3': '+++++++++value3'}
```

CHAPTER 11

Removing the oldest revision of the traceable dict

This option allows the user to contol the amount of revisions stored in the traceable-dict object, by trimming the tail of the trace stored in the traceable-dict. The oldest revision is cleared out and cannot be returned to again.

```
>>> from traceable_dict import TraceableDict
>>>
>>> d1 = {'old_key': 'old_value'}
>>> d2 = {'old_key': 'new_value'}
>>> d3 = {'old_key': 'even_newer_value'}
>>>
>>> D1 = TraceableDict(d1)
>>> D1.commit(revision=1)
>>>
>>> D1 = D1 | d2
>>> D1.commit(revision=2)
>>>
>>> D1 = D1 | d3
>>> D1.commit(revision=3)
>>>
>>> D1.revisions
[1, 2, 3]
>>> D1.remove_oldest_revision()
>>> D1. revisions
[2, 3]
```