learning-utility

Release 0.1.5

Dec 05, 2019

Contents:

learning-utility is a package of utilities for small-scale machine learning tasks with scikit-learn.

CHAPTER 1

Installation

```
pip install Lutil
```

CHAPTER 2

Key Features

2.1 Cache Intermediate Results

`InlineCheckpoint` can cache the computation result in the first call. Since then, if nothing has changed, it retrieves the cache and skips computation.

Suppose you have such a .py file.

```
from Lutil.checkpoints import InlineCheckpoint

a, b = 1, 2
with InlineCheckpoint(watch=["a", "b"], produce=["c"]):
   print("Heavy computation.")
   c = a + b

print(c)
```

Run the script, you will get:

```
Heavy computation.
3
```

Run this script again, the with-statement will be skipped. You will get:

```
3
```

Once a value among `watch` changes or the code inside the with-statement changes, re-calculation takes place to ensure the correct output.

Please check out the details for InlineCheckpoint.

2.2 Save Prediction Result According to the Given Format

Lots of machine learning competitions require a .csv file in a given format. Most of them provide an example file.

In example.csv:

```
id, pred
1, 0.25
2, 0.45
3, 0.56
```

Run:

```
>>> import numpy as np
>>> from Lutil.dataIO import AutoSaver

>>> result = np.array([0.2, 0.4, 0.1, 0.5])
        # Typical output of a scikit-learn predictor

>>> ac = AutoSaver(save_dir="somedir", example_path="path/to/example.csv")
>>> ac.save(result, "some_name.csv")
```

Then in your somedir/some_name.csv:

```
id, pred
1, 0.2
2, 0.4
3, 0.1
4, 0.5
```

It also works if the `result` is a pandas DataFrame, Series, 2-dim numpy array, etc. Also, the encoding, seperator, header, index of the example.csv will all be recognized.

Please check out the details for AutoSaver.

2.2.1 checkpoints: Cache Intermediate Results

There are usually some intermediate results when doing machine learning tasks. For example, the data after preprocessing. This module is useful for caching them on the disk, and skip re-calculation when is called afterwards.

Note: The cache files will be stored in your ./.Lutil-checkpoint directory, if you want to clean the cache, just delete it. You might also want to add this directory to your .gitignore.

Contents

InlineCheckpoint, the Context Manager

`InlineCheckpoint` is a context manager. It caches the output produced within the with-statement. When the script is executed later with the same condition, the cache is retrieved thus avoiding re-computation.

It is fully compatible with the jupyter notebook, and is often useful when using it for machine learning.

class Lutil.checkpoints.**InlineCheckpoint**(*, *watch*, *produce*)

Parameters

- **watch** (*list or tuple*) – List of names of variables used to identify a computing context
- **produce** (*list or tuple*) – List of names of variables whose values are generated within the with-statement

Basic Example

We have provided the simplest example in the welcome page, here is a more practical one.

Suppose you have such a .py file.

```
from Lutil.checkpoints import InlineCheckpoint

import numpy as np
from sklearn.decomposition import PCA

np.random.seed(0)                    # If random seed is not set, data will change␣
↪every time
data = np.random.randn(10000, 1000) # Simulate a large dataset
pca = PCA(20)                        # A typical scikit-learn transformer

with InlineCheckpoint(watch=["data", "pca"], produce=["data_t"]):
    data_t = pca.fit_transform(data)
    print("A thousand years later.")

print(data_t.shape)
```

Run the script, you will get:

```
A thousand years later.
(10000, 20)
```

Run this script again, the with-statement will be skipped. But the `produce` will be retrieved from cache, you will get:

```
(10000, 20)
```

Condition of Re-computation

If the variables or objects you watched have changed, for example, the data or the parameter for the scikit-learn transformer, code in the with-statement will be executed to retrieve the correct result.

For instance, if you replace `pca = PCA(20)` with `pca = PCA(50)` and run the script again, you will get:

```
A thousand years later.
(10000, 50)
```

The code in the with-statement is also monitored to detect whether the condition has changed.

For example, if you replace `data_t = pca.fit_transform(data)` with `data_t = data` The re-computation will be executed:

```
A thousand years later.
(10000, 1000)
```

Thus, please make sure that everything affecting the computation result is included in the `watch`.

Important: If you are watching the transformation result of a scikit-learn transformer. Make sure to set the transformer's `random_state` parameter if it has one. Otherwise it will vary each time, and be considered a different computation context. Thus the with-statement will not be skipped.

Format of Watch and Produce

Basically, the items in the `watch`/`produce` list shoule be valid variable names in Python.

The `watch` and `produce` can also be attributes of some object, using the `.` syntax.

This works:

```
class Foo: pass

f = Foo()
f.a = 1

with InlineCheckpoint(watch=["f.a"], produce=["f.b"]):
    f.b = f.a
print(f.b)
```

However, the slice syntax is not yet supported. This will cause error:

```
d = {'a':1}

with InlineCheckpoint(watch=["d['a']"], produce=["d['b']"]):
    d['b'] = d['a']
```

Caution: Because of some limitation of python magic we used to skip the code block and load the cached data, InlineCheckpoint **to produce variables** is **not supported within a function or method**.

This will not work!

```
def func(a):
    with InlineCheckpoint(watch=["a"], produce=["b"]):
        b = a
    return b
```

However, producing attributes of an object works well:

```
def func(a):
    f = Foo()
    with InlineCheckpoint(watch=["a"], produce=["f.b"]):
        f.b = a
    return f.b
```

Nevertheless, *checkpoint as a decorator* is recommended for a function. Besides, if you use this, the return statement should not be included in the with-statement.

Watching a Complex Object

If the object you are watching has some attributes, which are neither basic data types nor pd.DataFrame/np.ndarray, a warning will be raised. It is not recommended to do so. Instead, explicitly watch those attributes which affects the computation, using the `.` syntax.

```
class Bar: pass
f = Foo()
f.bar = Bar()

with InlineCheckpoint(watch=["f"], produce=["f.a"]):
    f.a = 1
```

will give you:

```
ComplexParamsIdentifyWarning: A complicated object is an attribute of <__main__.Foo␣
↪object at 0x000001CE66E897B8>,
it may cause mistake when detecting whether there is checkpoint for this call.
```

checkpoint, the Decorator

`checkpoint` is a decorator which cache the return value of a function or method on the disk. When the function is called later with the same condition, retrieve the cached value and return, avoiding re-computation.

`@Lutil.checkpoints.`**`checkpoint`**

`@Lutil.checkpoints.`**`checkpoint`**(*ignore=[]*)

Parameters **`ignore`** (*`list or tuple`*) – Optional, list of names of variables ignored when identifying a computing context

Basic Example

Suppose you have such a .py file:

```
from Lutil.checkpoints import checkpoint

@checkpoint
def foo(a, b):
    print("Heavy computation.")
    return a + b

print(foo(1, 2))
print(foo(1, 2))
```

Run this script, you will get:

```
Heavy computation.
3
3
```

In the second call of `foo`, the computation is skipped, and the return value is retrieved from cache.

In machine learning tasks, the parameters are often pd.DataFrame or np.ndarray, `checkpoint` works well on them.

Condition of Re-computation

If the parameter of the function have changed, the function will be called again to retrieve the correct result.

In the previous example, add a new function call

```
print(foo(1, 3))
```

You will get:

```
Heavy computation.
4
```

If the code of the function have changed, re-computation takes place as well.

In the previous example, change the function definition from `return a + b` to `return a - b`, and call `print(foo(1, 2))` again, you will get:

```
Heavy computation.
-1
```

Important: If some parameter of the decorated function is the transformation result of a scikit-learn transformer. Make sure to set the transformer's `random_state` parameter if it has one. Otherwise it will vary each time, and be considered a different computation context. Thus the decorated function will not be skipped.

Force Recomputation

Sometimes you want to arbitrarily rerun a function and recompute the result, probably because some files you read inside the function is changed, which requires recompute but can not be identified by default. You can add `__recompute__=True` parameter to the function call to force recomputation.

```
print(foo(1, 2, __recompute__=True))
print(foo(1, 2, __recompute__=True))
```

You will get:

```
Heavy computation.
3
Heavy computation.
3
```

The second function call is forced to recompute.

Ignore Some Parameters

By default, `checkpoint` monitors all the parameters of the decorated function. However, the `ignore` parameter can be used if some of them are not contributing to the return value.

```
@checkpoint(ignore=["useless"])
def bar(a, useless):
    print("Runned.")
    return a + 1

print(bar(1, True))
print(bar(1, False))
```

Althought the value of `useless` have changed, there will be no re-computation. You will get:

```
Runned.
2
2
```

Complex Object as a Parameter

If some parameters of the decorated function are neither basic data types nor pd.DataFrame/np.ndarray, a warning will be raised. It is not recommended to do so.

```
@checkpoint
def func(foo):
    return foo

class Foo: pass
f = Foo()
f.foo = Foo()

func(foo)
```

You will get:

```
ComplexParamsIdentifyWarning: A complicated object is an attribute of <__main__.Foo␣
↪object at 0x00000224A1575358>,
it may cause mistake when detecting whether there is checkpoint for this call.
```

See Also

joblib.Memory is similar to our `checkpoint` decorator. It is more powerful, while ours is more concise.

However, `joblib` is not providing anything similar to our `InlineCheckpoint`, while this is often necessary in some jupyter notebook based solutions. This is also the motivation of this module.

Another important difference is that, if the code of the function changes, `joblib.Memory` only caches the result of the latest function version.

```
from joblib import Memory
memory = Memory("dir", verbose=0)

@memory.cache
def f(x):
    print('Running.')
    return x

f(1)
```

Run this, you get:

```
Running.
```

If you change `print('Running.')` to `print('Running again.)'`, you will get:

```
Running again.
```

Now, if you change it back to `print('Running')`, it will not retrieve the result in the first run. Instead, the computation happens again:

```
Running.
```

However, if you are using our `checkpoint`.

```
from Lutil.checkpoints import checkpoint

@checkpoint
def f(x):
    print('Running.')
    return x
```

Do the similar thing, and in the third run, the computation will be skipped. The result in the first run will be retrieved.

2.2.2 dataIO: Read Data, Save Result

Contents

AutoSaver, Auto-format and Save Prediction Results

In some machine learning competitions, like kaggle, a .csv file is required for result submission. Its format is usually illustrated in an example file. `AutoSaver` inspects the required format from the example, and save your result.

class `Lutil.dataIO.`**AutoSaver**(*save_dir=""*, *example_path=None*, ***default_kwargs*)

Parameters

- **save_dir** (*str*) – Directory where your results will be saved
- **example_path** (*str*) – Optional, path to the example .csv file
- **default_kwargs** – Default keyword arguments arbitrarily used for *DataFrame.to_csv()*

`Autosaver.`**save**(*self*, *X*, *filename*, *memo=None*, ***kwargs*)

Parameters

- **X** (*pd.DataFrame, pd.Series or np.ndarray*) – The prediction result to be saved
- **filename** (*str*) – Optional, the filename of the result file. Will use `datetime.datetime.now().strftime(r"%m%d-%H%M%S") + ".csv"` if left empty
- **memo** (*str*) – Optional, the memo logged for this result
- **kwargs** – Other keyword arguments arbitrarily used for *DataFrame.to_csv()*

Auto-format Examples

We have provided the simplest example in the welcome page. More format and data types can be inferred.

For example, if the index in the example starts from zero and there is no headers

```
0, 0.1
1, 0.1
2, 0.1
```

Run:

```
>>> import numpy as np
>>> from Lutil.dataIO import AutoSaver

>>> result = np.array([0.2, 0.4, 0.1, 0.5])
        # Typical output of a scikit-learn predictor

>>> ac = AutoSaver(save_dir="somedir", example_path="path/to/example.csv")
>>> ac.save(result, "some_name.csv")
```

Then in your somedir/some_name.csv:

```
0, 0.2
1, 0.4
2, 0.1
3, 0.5
```

Some competitions use a hash-like value as the index. For instance, in your example.csv:

```
hash, value
aaffc2, 0.1
spf2oa, 0.1
as2nw2, 0.1
```

Then you should include the index in the `X` parameter. However, how you achieve this is quite at will.

```
>>> import numpy as np
>>> import pandas as pd
>>> from Lutil.dataIO import AutoSaver

>>> index = ["aaffc2", "spf2oa", "as2nw2", "wn2ajn"]
>>> pred = np.array([0.2, 0.4, 0.1, 0.5])

>>> # In either of the four ways:
>>> result = pd.Series(pred, index=index)
>>> result = pd.DataFrame({
...     "ix": index,
...     "pred": pred
... })
>>> result = pd.DataFrame({"pred":pred}, index=index)
>>> result = np.array([index, pred]).T

>>> ac = AutoSaver(save_dir="somedir", example_path=r"explore\doctests\example.csv")
>>> ac.save(result, "some_name.csv")
```

In your somedir/some_name.csv, the results will be perfectly saved:

```
hash,value
aaffc2,0.2
spf2oa,0.4
as2nw2,0.1
wn2ajn,0.5
```

As long as the object you are saving is a numpy.ndarray or a pd.Series/pd.DataFrame, and it "looks like" the final csv file according to the example, the auto-format will work.

Log Memo for the Results

Sometimes you would like a memo, a description for the results you have saved. Then you can use the `memo` parameter in the `AutoSaver.save` method.

```
>>> result1 = np.array([0.2, 0.4, 0.1, 0.5])
>>> result2 = np.array([0.2, 0.3, 0.1, 0.6])

>>> ac = AutoSaver(save_dir="somedir", example_path="path/to/example.csv")
```

(continues on next page)

(continued from previous page)

```
>>> ac.save(result1, "result1.csv", memo="Using Random Forest.")
>>> ac.save(result2, "result2.csv", memo="Using XGBoost.")
```

Then you will find this in your somedir/memo.txt:

```
result1.csv: Using Random Forest.
result2.csv: Using XGBoost.
```

All the new memos will be appended to the end of memo.txt.

Arbitrarily Using Keyword Arguments

If the format in your example.csv is too complex and `AutoSaver` failed to inspect that, you can also pass a pandas.Series or pandas.DataFrame to the `save` method, and arbitrarily assign arguments to use its `to_csv` method.

It is true that this is not very meaningful, comparing with directly calling `DataFrame.to_csv`, except that it gives you the access to our "memo" feature, and only have to set the parameters once while saving multiple results.

For example:

```
>>> df = pd.DataFrame({
...     "ix":[1,2,3],
...     "pred":[0.1,0.2,0.3]
... })

>>> ac = AutoSaver(save_dir="somedir", index=False)
>>> ac.save(df, "result1.csv")
```

This is equivalent to:

```
>>> df.to_csv("somedir/result1.csv", index=False)
```

You can also add more arguments when calling `save`:

```
>>> ac.save(df, "result2.csv", header=True)
```

Both the keyword arguments assigned when initializing and when calling `save` will be applied, which is equivalent to:

```
>>> df.to_csv("somedir/result2.csv", index=True, header=True)
```

When you use arbitrary arguments, you cannot use the `example_path` feature. They contradicts each other.

DataReader, Raw Data Management

If you want to read the dataset multiple times or across modules, it can be boring to copy-paste your `pd.read_csv()` statement. `DataReader` is a dataset manager which allows you to set the reading parameter only once, and get the dataset anytime after without more effort.

class Lutil.dataIO.**DataReader**(*train_path=None*, *test_path=None*, *val_path=None*, *_id="default"*, *read_func=None*, ***read_kwargs*)

Parameters

- **train_path** (*str*) – Optional, path to the train set

- **test_path** (*str*) – Optional, path to the test set
- **val_path** (*str*) – Optional, path to the validation set
- **_id** (*str*) – Optional, identifier for multiple datasets
- **read_func** (*callable*) – Optional, function used for reading data, default `pd.read_csv`
- **read_kwargs** – Other keyword arguments for applying to the `read_func`

`DataReader.`**train**(*self*)
Returns the train set.

`DataReader.`**test**(*self*)
Returns the test set.

`DataReader.`**val**(*self*)
Returns the validation set.

Basic Examples

By default, `pandas.read_csv` will be used to read csv datasets, whose path are assigned when initializing the `DataReader` object. You can also assign the parameters for `read_csv` when initializing.

```
>>> from Lutil.dataIO import DataReader

>>> reader = DataReader("path/to/train.csv",
...                     "path/to/test.csv",
...                     "path/to/val.csv", index_col=1)

>>> train = reader.train()
```

This is equivalent to:

```
>>> train = pd.read_csv("path/to/train.csv", index_col=1)
```

Likewise, you can also call

```
>>> test = reader.test()
>>> val = reader.val()
```

which are equivalent to:

```
>>> test = pd.read_csv("path/to/test.csv", index_col=1)
>>> val = pd.read_csv("path/to/val.csv", index_col=1)
```

Ever since you have initialized one instance, you can completely forget about the object and all parameter configurations. In the same runtime, even in other files, this will be able to retrieve the train set as before.

```
>>> DataReader().train()
```

It is the same for the test set and the validation set.

Accessing Multiple Datasets

Most small-scale machine learning tasks only have one dataset, which is our basic usage. However, if you want to access multiple datasets, you can assign the `_id` parameter. This will work accross files as well.

```
>>> DataReader("path/to/train_1.csv", _id="1", index_col=1)
>>> DataReader("path/to/train_2.csv", _id="2", nrows=500)

>>> train_1 = DataReader(_id="1").train()
>>> # Equivalent to
>>> train_1 = pd.read_csv("path/to/train_1.csv", index_col=1)

>>> train_2 = DataReader(_id="2").train()
>>> # Equivalent to
>>> train_2 = pd.read_csv("path/to/train_2.csv", nrows=500)
```

Using other Reading Function

If the data source is not a csv file, and you want to read them with other functions, you can pass a callable to the `read_func` parameter.

```
>>> import pandas as pd
>>> reader = DataReader("path/to/train.json", read_func=pd.read_json)
>>> train = reader.train()
```

This is equivalent to:

```
>>> train = pd.read_json("path/to/train.json")
```

Applying other keyword parameter is the same as before, pass them when initializing the `DataReader` object and it will be passed when actually calling the `read_func`.

As you see, this will only work if the dataset is stored in one file, and the `read_func` take the path as the first parameter.

2.2.3 Changelog

v0.1.5

- Bug fixes
- Now the file name of `Autosaver.save` is optional, will use `datetime.datetime.now().strftime(r"%m%d-%H%M%S") + ".csv"` if left empty
- Add `__recompute__` functionality for `checkpoint`

v0.1.4

- Fix requirements.txt for Windows environment

v0.1.3

- **Add support for two `AutoSaver` cases**
  - Fix: when a column header in the example file is empty, the output csv will no longer have an `Unnamed: 0` as header.
  - Feature: now it can handle ruleless indexes, as long as the example csv and the to-submit object has the same number of rows

v0.1.2

- Fix dependencies in requirements.txt and setup.py
- Use better way to hash a pandas object

v0.1.1

- Fix changelog in the documentation (v0.0.1 -> v0.1.0)
- Fix fields in the setup.py
- More badges in the readme and documentation
- No code change

v0.1.0

- **Initialize the `checkpoints` module**
  - `InlineCheckpoint`
  - `checkpoint`
- **Initialize the `dataIO` module**
  - `AutoSaver`
  - `DataReader`

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index