chariots Documentation

Release 0.2.4

Antoine Redier

Jun 09, 2021

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Stable release

To install chariots, run this command in your terminal:

```
$ pip install chariots
```

This is the preferred method to install chariots, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for chariots can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/aredier/chariots
```

Or download the tarball:

```
$ curl  -OL https://github.com/aredier/chariots/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

TWO

GENERAL PRINCIPLES

2.1 Versioning

One of the key principles of Chariots is versioning. This means that Chariots will enforce what we consider to be good versioning practices during the deployment, retraining and updating phases over the lifetime of your your *Chariots* app.

2.1.1 Why Version Machine Learning Pipelines?

You might ask yourself why we would need to version the different models in our ML pipelines. For Kaggle competition I just train my models in order predict on the test set and submit right? Although this workflow works to a certain extent on small production project it can soon become quite a mess.

For instance if you try to build an NLP micro-service in your architecture, you will probably have a unique well performing word embedding model and several other smaller models (intent classifier, POS, . . .) that you retrain more often. In this setup you have two choices when it comes to training:

- you can retrain every model in the right order every time you redeploy your micro-service. This is the extension of the Kaggle approach but means you will end up loosing some time retraining unnecessary models (which will slow down your deployment process and cost you in machine time)
- you can trust yourself to know what models need to be retrained and to do it right each time. If you choose to do this you will eventually end up in a mess where you forgot to retrain a classifiers after retraining your embedding model (and your classifier outputting nonsense . . .)

Chariots provides you with a third option by enforcing the versioning logic in your pipelines framework. This means that when you try to load (at start up or after a manual retrain) a pipeline, *Chariots* will check that every model has a valid link (has been trained with) to the version of the preceding model and will not load if said valid link is not found

2.1.2 Semantic Versioning in Chariots

Chariots tries to stick to the Semantic Versioning guidelines. This means that all the versions in *Chariots* are comprised of three subversions (Major, Minor, Patch). This also means that the checks chariots makes on versions (described above) will only apply to the major versions (although we plan to make this user customizable in the future).

One major difference between Chariots and traditional Semantic versioning is the use of incremental number. For practical reasons, chariots uses three hashes instead of thee numbers and the ordering of those versions comes from the time stamp of the version.

2.1.3 Version and Pipeline Interactions

In this section we will try to explain in more details how *chariots* creates and updates links between different versions of your models.

Chariots treats each pipeline as a DAG using part of some shared nodes. if we take back our NLP example:

here a link between to nodes of a pipeline represents a valid version meaning that that here all the nodes accept their parent node in the pipeline. However if we retrain our embeddings, our DAGs will look like this:

here there is no valid link between the embedding and the other models (POS and Intent). We than need to retrain and recreate those links:

Once this is done a new version of our POS and intent models have been created and a valid link has been submitted by the training pipelines. when trying to reload the prediction pipelines, those will see this new link and accept the new versions of their nodes.

2.2 Pipelines, Nodes & Ops

The *Chariots* framework is built around three main types that we use to build a *Chariots* server: Pipelines, Nodes and Ops. In this article we will go over those three main building blocks in general terms. You can of course check the *API documentation* to check how to use them technically.

2.2.1 Ops

Ops are the atomic computational unit in the *Chariots* framework, meaning that they are part of a more complete pipeline that couldn't (or at least it wouldn't make sense to) be divided in smaller chunks of instruction. Ops are actually the only types that are versioned in the framework (also nodes have versions that are derived from ops).

For instance a machine learning model will be an Op versioned according to it's several parameters and it's last training time.

Also ops have requirements (in terms of number and types that they receive as arguments to their *execute* method) they are treated as agnostic from the pipeline they are called in (an op can be used in multiple pipelines for instance)

2.2.2 Nodes

Nodes represent a slot in a pipeline meaning. They define the interactions within the pipelines by connecting to their upstream and downstream node(s). Nodes can be built upon Ops (*Node(my_op)*) but not necessarily, for instance DataNodes are opless nodes, moreover ABTesting nodes (feature of the upcoming 0.3 release) would nodes using multiple ops. Another example wound be to use whole pipelines to execute the slot of a node (*Node(my_pipeline*)

For instance a node would represent the preprocessing slot in a pipeline that you could use Op1 or Op2 to fill that slot (or both in the case of ABTesting).

2.2.3 Pipelines

Pipelines are callable collections of nodes that are exposed to your users through the *Chariots* server (you can also use them directly but it is not the recommended way of using them). They can also be used inside other pipelines to fill in a specific node's slot.

In these few pages, we will discuss the guiding principles and general logic behind the *Chariots* development. These will be theoretical guides to understand the major rationale behind the technical decisions taken in the framework.

As the framework is still in its early stage and evolving quite rapidly, these principles are not yet fully implemented.

CHAPTER

THREE

TUTORIALS

We have provided a (more coming . . .) basic tutorial to demonstrate the basics of the *Chariots* framework and get you going quickly.

3.1 Iris Tutorial

In this beginners tutorial we will build a small *Chariots* server to serve predictions on the famous iris dataset. If you want to see the final result, you can produce it directly using the *chariots new* command (see the *chariots template* for more info).

before starting, we will create a new project by calling the *new* command in the parent directory of where we want our project to be and leave all the default options

```
chariots new
```

3.1.1 Ops

we first need to design the individual ops we will build our pipelines from.

Data Ops

First we will need an op that downloads the dataset, so in *iris/ops/data_ops/download_iris.py*

```
>>> import pandas as pd
>>> from chariots.base import BaseOp
>>> from sklearn import datasets
...
...
>>> class DownloadIris(BaseOp):
...
...     def execute(self):
...         iris = datasets.load_iris()
...         df = pd.DataFrame(data=iris['data'], columns=iris['feature_names'])
...         df["target"] = iris["target"]
...         return df
```

Machine Learning Ops

we will than need to build our various machine learning ops. For this example we will be using a PCA and than a Random Forest in our pipeline. We will place those ops in the *iris.ops.model_ops* subpackage

```
>>> from sklearn.decomposition import PCA
>>> from chariots.versioning import VersionType, VersionedFieldDict
>>> from chariots.sklearn import SKUnsupervisedOp
...
...
>>> class IrisPCA(SKUnsupervisedOp):
...
...     model_class = PCA
...     model_parameters = VersionedFieldDict(
...         VersionType.MAJOR,
...         {
...             "n_components": 2,
...         }
...     )
```

```
>>> from chariots.versioning import VersionType, VersionedFieldDict
>>> from chariots.sklearn import SKSupervisedOp
>>> from sklearn.ensemble import RandomForestClassifier
...
...
>>> class IrisRF(SKSupervisedOp):
...
...     model_class = RandomForestClassifier
...     model_parameters = VersionedFieldDict(VersionType.MINOR, {"n_estimators": 5,
↪"max_depth": 2})
```

Preprocessing Ops

we will not be using preprocessing ops per say but we will need an op that splits our saved dataset between X and y as otherwise we will not be able to separate the two.

```
>>> from chariots.base import BaseOp
...
...
>>> class XYSplit(BaseOp):
...
...     def execute(self, df):
...         return df.drop('target', axis=1), df.target
```

3.1.2 Pipelines

We will than need to build our pipelines using the nodes we have just created:

Data Pipelines

We have our op that downloads the dataset. We than need to feed this dataset into a data saving node that will persist it for future uses (as the iris dataset is quite light, we could wire the download directly into the training pipeline but we will persist it to demonstrate that dynamic).

```
>>> from chariots import Pipeline
>>> from chariots.nodes import DataSavingNode, Node
>>> from chariots.serializers import CSVSerializer
...
...
>>> download_iris = Pipeline(
...     [
...         Node(DownloadIris(), output_nodes="iris_df"),
...         DataSavingNode(serializer=CSVSerializer(), path="iris.csv",
...                        input_nodes=["iris_df"])
...     ], "download_iris"
... )
```

Machine Learning Pipelines

Once we have our data set saved, we will need to use it to train our models, we will than create a training pipeline:

```
>>> from chariots import MLMode, Pipeline
>>> from chariots.nodes import DataLoadingNode, Node
>>> from chariots.serializers import CSVSerializer
...
...
>>> train_iris = Pipeline(
...     [
...         DataLoadingNode(serializer=CSVSerializer(), path="iris.csv",
...                         output_nodes="iris"),
...         Node(XYSplit(), input_nodes=["iris"], output_nodes=["raw_X", "y"]),
...         Node(IrisPCA(MLMode.FIT_PREDICT), input_nodes=["raw_X"],
...              output_nodes="pca_X"),
...         Node(IrisRF(MLMode.FIT), input_nodes=["pca_X", "y"])
...     ], "train_iris"
... )
```

Once the models will be trained, we will need to provide a pipeline for serving our models to our users. To do so, we will create a pipeline that takes some user provided values (raws of the iris format) and retruns a prediction to the user:

```

```

3.1.3 App & Client

Once our pipelines are all done, we will only need to create *Chariots* server to be able to serve our pipeline:

```
>>> from chariots import Chariots
...
...
>>> app = Chariots(
...     [download_iris, train_iris, pred_iris],
...     path=app_path,
...     import_name="iris_app"
... )
```

Once this is done we only need to start our server as we would with any other *Flask'app (the 'Chariots* type inherits from the *Flask* class). For instance using the cli in the folder containing our *app.py*:

```
flask
```

our server is now running and we can execute our pipelines using the chariots client:

```
>>> from chariots import Client
...
...
>>> client = Client()
...
```

we will need to execute several steps before getting to a prediction:

- download the dataset
- train the operations
- save the trained machine learning ops
- reload the prediction pipeline (to use the latest/trained version of the machine learning ops)

```
>>> client.call_pipeline(download_iris)
>>> client.call_pipeline(train_iris)
>>> client.save_pipeline(train_iris)
>>> client.load_pipeline(pred_iris)
...
>>> client.call_pipeline(pred_iris, [[1, 2, 3, 4]])
[1]
```

CHAPTER

FOUR

API DOCS

4.1 chariots.base

The Base Module Gathers all the main classes in the Chariots framework that can be subclassed to create custom behaviors:

- creating new Ops for preprocessing and feature extraction (subclassing *BaseOp*)
- supporting new ML frameworks with *BaseMLOp*
- creating a custom node (ABTesting, ...) with the *BaseNode*
- changing the execution behavior of pipelines (Multiprocessing, cluster computing, ...) with *BaseRunner*
- saving your ops and metadata to a different cloud provider with *BaseSaver*
- creating new serialisation formats for datasets and models with *BaseSerializer*

class chariots.base.**BaseOp**(*op_callbacks: Optional[List[chariots.callbacks._op_callback.OpCallBack]] = None*)

Bases: object

The ops are the atomic computation units of the Chariots framework. Whereas a *Node* represents a slot in a pipeline and the interactions between that spot and the rest of the pipeline, the op will actually be doing the computation.

To subclass the *BaseOp* class and create a new Op, you need to override the *execute* method:

```
>>> class AddOp(BaseOp):
...     number_to_add = 1
...
...     def execute(self, op_input):
...         return op_input + self.number_to_add
```

and then you can execute the op alone:

```
>>> AddOp().execute(3)
4
```

or within a pipeline (that can be deployed)

```
>>> pipeline = Pipeline([Node(AddOp(), ["__pipeline_input__"], "__pipeline_output_
↪_")], "simple_pipeline")
>>> runner.run(pipeline, 3)  # of course you can use a `Chariots` server to serve␣
↪our pipeline and op(s)
4
```

The *BaseOp* class is a versioned class (see the *versioning* module for more info) so you can use *VersionedField* with it

```
>>> class AddOp(BaseOp):
...     number_to_add = VersionedField(3, VersionType.MAJOR)
...
...     def execute(self, op_input):
...         return op_input + self.number_to_add


>>> AddOp.__version__
<Version, major:36d3c, minor: 94e72, patch: 94e72>
>>> AddOp.number_to_add
3
```

and changing the field will change the version:

```
>>> class AddOp(BaseOp):
...     number_to_add = VersionedField(4, VersionType.MAJOR)
...
...     def execute(self, op_input):
...         return op_input + self.number_to_add


>>> AddOp.__version__
<Version, major:8ad66, minor: 94e72, patch: 94e72>
```

Parameters **op_callbacks** – *OpCallbacks objects* to change the behavior of the op by executing some action before or after the op'execution

after_execution(*args: List[Any]*, *output: Any*) → Any
method used to create a one-off (compared to using a *callback*) custom behavior that gets executed after the the op itself

Parameters

- **args** – the arguments that were passed to the op
- **output** – the output of the op

property allow_version_change
whether or not this op accepts to be loaded with the wrong version. this is usually False but is useful when loading an op for retraining

before_execution(*args: List[Any]*)
method used to create a one-off (compared to using a *callback*) custom behavior that gets executed before the the op itself

Parameters **args** – the arguments that are going to be passed to the operation

execute(**args*, ***kwargs*)
main method to override. it defines the behavior of the op. In the pipeline the argument of the pipeline will be passed from the node with one argument per input (in the order of the input nodes)

execute_with_all_callbacks(*args*)
executes the op itself alongside all it's callbacks (op callbacks and *before/after_execution* methods)

Parameters **args** – the arguments to be passed to the *execute* method of the op

Returns the result of the op

property name
the name of the op. this is mainly use to find previous versions and saved ops of this op in the op_store

property op_version
the version the op uses to pass to the pipeline to identify itself. This differs from the *__version__* method in that it can add some information besides the class Fields (for instance last training time for ML Ops)

class chariots.base.**BaseMLOp**(*mode: chariots._ml_mode.MLMode*, *op_callbacks: Optional[List[chariots.callbacks._op_callback.OpCallBack]] = None*)
Bases: chariots.ops._loadable_op.LoadableOp

an BaseMLOp are ops designed specifically to be machine learning models (whether for training or inference). You can initialize the op in three distinctive *ml mode*:

- *FIT* for training the model
- *PREDICT* to perform inference
- *FIT_PREDICT* to do both (train and predict on the same dataset

the usual workflow is to a have a training and a prediction pipeline. and to:

- execute the training pipeline:
- save the training pipeline
- reload the prediction pipeline
- use the prediction pipeline

here is an example:

first create your pipelines:

```
>>> train = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.FIT_PREDICT), input_nodes=["x"], output_nodes="x_
↪transformed"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train')

>>> pred = Pipeline([
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["__pipeline_input__"], output_
↪nodes="x_transformed"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes=['__pipeline_output__'])
... ], 'pred')
```

and then to train your pipelines and make some predictions:

```
>>> client.call_pipeline(train)
>>> client.save_pipeline(train)
>>> client.load_pipeline(pred)
>>> client.call_pipeline(pred, [[1, 2, 3, 4]])
[1]
```

If you want to create a new MLOp class (to accommodate an unsupported framework for instance), you need to define:

- how to fit your op with the *fit* method
- how to perform inference with your op with the *predict* method

- define how to initialize a new model with the *_init_model* method

and eventually you can change the *serializer_cls* class attribute to change the serialization format of your model

Parameters **op_callbacks** – *OpCallbacks objects* to change the behavior of the op by executing some action before or after the op's execution

property allow_version_change
whether or not this op accepts to be loaded with the wrong version. this is usually False but is useful when loading an op for retraining

execute(**args*, ***kwargs*)
executes the model action that is required (train, test or both depending in what the op was initialized with

abstract fit(**args*, ***kwargs*)
fits the inner model of the op on data (in args and kwargs) this method must not return any data (use the FIT_PREDICT mode to predict on the same data the op was trained on)

load(*serialized_object: bytes*)
Receives serialize bytes of a newer version of this class and sets the internals of he op accordingly.

Parameters **serialized_object** – the serialized bytes of this op (as where outputed by the *serialize* method

property mode
the mode this op was instantiated with

property op_version
the version the op uses to pass to the pipeline to identify itself. This differs from the *__version__* method in that it can add some information besides the class Fields (for instance last training time for ML Ops)

abstract predict(**args*, ***kwargs*) → Any
the method used to do predictions/inference once the model has been fitted/loaded

serialize() → bytes
serializes the object into bytes (to be persisted with a Saver) to be reloaded in the future (you must ensure the compatibility with the *load* method

Returns the serialized bytes representing this operation

serializer_cls
alias of `chariots.serializers._dill_serializer.DillSerializer`

training_update_version = 'patch'

class chariots.base.**BaseRunner**
Bases: `abc.ABC`

a runner is used to define the execution behavior of a Pipeline. there main entry point is the *run* method

```
>>> runner.run(is_odd_pipeline, 3)
True
```

To create a new runner (for instance to execute your pipeline on a cluster) you only have to override *run* method and use the *Pipeline*'s class methods (for instance you might want to look at *extract_results*, *execute_node*)

abstract run(*pipeline: chariots._pipeline.Pipeline*, *pipeline_input: Optional[Any] = None*)
runs a pipeline, provides it with the correct input and extracts the results if any

Parameters

- **pipeline** – the pipeline to run
- **pipeline_input** – the input to be given to the pipeline

Returns the output of the graph called on the input if applicable

class chariots.base.**BaseSaver**(*root_path: str*)
Bases: abc.ABC

abstraction of a file system used to persist/load assets and ops this can be used on the actual local file system of the machine the *Chariots* server is running or on a bottomless storage service (not implemented, PR welcome)

To create a new Saver class you only need to define the *Save* and *Load* behaviors

Parameters **root_path** – the root path to use when mounting the saver (for instance the base path to use in the the file system when using the *FileSaver*)

load(*path: str*) → bytes
loads the bytes serialized at a specific path

Parameters **path** – the path to load the bytes from.You should not include the *root_path* of the saver in this path: loading to */foo/bar.txt* on a saver with */my/root/path* as root path will load */my/root/path/foo/bar.txt*

Returns saved bytes

save(*serialized_object: bytes*, *path: str*) → bool
saves bytes to a specific path.

Parameters

- **serialized_object** – the bytes to persist
- **path** – the path to save the bytes to. You should not include the *root_path* of the saver in this path: saving to */foo/bar.txt* on a saver with */my/root/path* as root path will create/update */my/root/path/foo/bar.txt*

Returns whether or not the object was correctly serialized.

class chariots.base.**BaseSerializer**
Bases: abc.ABC

serializers are helper classes for communication and persistence through out the *Chariots* framework. There mostly used by data nodes and and MLOps.

For instance if you want to make a pipeline that downloads the iris dataset splits it between train and test and use two different formats for the train and test (please don't . . .):

```
>>> save_train_test = Pipeline([
...     Node(IrisDF(), output_nodes='df'),
...     Node(TrainTestSplit(), input_nodes=['df'], output_nodes=['train_df',
↪'test_df']),
...     DataSavingNode(serializer=CSVSerializer(), path='/train.csv', input_
↪nodes=['train_df']),
...     DataSavingNode(serializer=DillSerializer(), path='/test.pkl', input_
↪nodes=['test_df'])
... ], "save")
```

fot MLOps if you want to change the default serialization format (for the model to be saved), you will need to change the *serializer_cls* class attribute

abstract deserialize_object(*serialized_object: bytes*) → Any
returns the deserialized object from serialized bytes (that will be loaded from a saver)

Parameters **serialized_object** – the serialized bytes

Returns the deserialized objects

abstract serialize_object(*target: Any*) → bytes
serializes the object into bytes (for ml ops *target* will be the model itself and not the op, for the data ops the *target* will be the input of the node)

Parameters **target** – the object that will be serialized

Returns the bytes of the serialized object

class chariots.base.**BaseNode**(*input_nodes: Optional[List[Union[str, BaseNode]]] = None, output_nodes: Union[List[str], str] = None*)

Bases: abc.ABC

A node represents a step in a Pipeline. It is linked to one or several inputs and can produce one or several ouptuts:

```
>>> train_logistics = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.FIT_PREDICT), input_nodes=["x"], output_nodes="x_
↪transformed"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')
```

you can also link the first and/or the last node of your pipeline to the pipeline input and output:

```
>>> pred = Pipeline([
...     Node(IrisFullDataSet(),input_nodes=['__pipeline_input__'], output_nodes=[
↪"x"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes=['__pipeline_output__'])
... ], 'pred')
```

Here we are showing the behavior of nodes using the *Node subclass* (used with ops).

If you want to create your own Node you will need to define the

- *node_version* property that gives the version of the node
- *name* property
- *execute* method that defines the execution behavior of your custom Node
- *load_latest_version* that defines how to load the latest version of this node

Parameters

- **input_nodes** – the input_nodes on which this node should be executed
- **output_nodes** – an optional symbolic name for the outputs of this node (to be used by downstream nodes in the pipeline. If this node is the output of the pipeline use *__pipeline_output__* or *ReservedNodes.pipeline_output*. If the output of the node should be split (for different downstream nodes to consume) use a list

check_version_compatibility(*upstream_node: chariots.base._base_nodes.BaseNode, store_to_look_in: chariots._op_store.OpStore*)
checks that this node is compatible with a potentially new version of an upstream node‘

Parameters

- **upstream_node** – the upstream node to check for version compatibality with

- **store_to_look_in** – the op_store to look for valid relationships between this node and upstream versions

Raises ***VersionError*** – when the two nodes are not compatible

abstract execute(**params*) → Any
executes the computation represented byt this node (loads/saves dataset for dataset nodes, executes underlyin op for *Node*

Parameters **params** – the inputs provided by the *input_nodes*

Returns the output(s) of the node

property is_loadable
whether or not this node can be loaded (this is used by pipelined to know which nodes to load

abstract load_latest_version(*store_to_look_in: chariots._op_store.OpStore*) → chariots.base._base_nodes.BaseNode
reloads the latest available version of thid node by looking for all available versions in the OpStore

Parameters **store_to_look_in** – the store to look for new versions and eventually for bytes of serialized ops

Returns this node once it has been loaded

abstract property name
the name of the node

abstract property node_version
the version of this node

property output_references
the different outputs of this nodes

persist(*store: chariots._op_store.OpStore, downstream_nodes: Optional[List[BaseNode]]*) → chariots.versioning._version.Version
persists this nodes's data (usually this means saving the serialized bytes of the inner op of this node (for the *Node* class

Parameters

- **store** – the store in which to store the node
- **downstream_nodes** – the node(s) that are going to accept the current version of this node as upstream

replace_symbolic_references(*symbolic_to_real_node: Mapping[str, NodeReference]*) → chariots.base._base_nodes.BaseNode
replaces all the symbolic references of this node: if an input_node or output_node was defined with a string by the user, it will try to find the node represented by this string.

Parameters **symbolic_to_real_node** – the mapping of all *NodeReference* found so far in the pipeline

Raises **ValueError** – if a node with multiple outputs was used directly (object used rather than strings)

Returns this node with all it's inputs and outputs as *NodeReferences* rather than strings

property require_saver
whether or not this node requires a saver to be executed this is usualy *True* by data nodes

property requires_runner
whether or not this node requires a runner to be executed (typically if the inner op is a pipelines)

4.2 chariots.callbacks

Callbacks are use to change the default behavior of an op or a pipeline in a reusable way, you can create callbacks to log performance or timing check output distribution or what ever you need around the pipeline or the ops execution.

There are two main types of callbacks:

- operation callbacks that give ou entry points before and after the execution of this specific op
- pipeline callback that give you entry points before and after the execution of the pipeline and in between each node

the order of execution of the callbacks are as follows:

- pipeline callbacks' *before_execution*
- pipeline callbacks' *before_node_execution* (for each node)
- op callbacks' *before_execution*
- op' *before_execution* method
- op's execute method
- op's *after_execution* method
- op callbacks' *after_execution*
- pipeline callbacks' *after_node_execution*

During the pipeline's execution, the inputs and outputs of the execution are being provided (when applicable), these are provided for information, DO NOT TRY TO MODIFY those (this is undefined behavior)

class chariots.callbacks.**OpCallBack**

Bases: object

an op callback is used to perform specific instructions at certain points around the operation's execution

to create your own op callback, you need to override either the *before_execution* or the *after_execution* method (or both)

```
>>> class PrintOpName(OpCallBack):
...
...     def before_execution(self, op: "base.BaseOp", args: List[Any]):
...         print('{} called with {}'.format(op.name, args))
```

```
>>> is_even_pipeline = Pipeline([
...     Node(AddOneOp(), input_nodes=['__pipeline_input__'], output_nodes=
↪'modified'),
...     Node(IsOddOp(op_callbacks=[PrintOpName()]), input_nodes=['modified'],
...          output_nodes=['__pipeline_output__'])
... ], 'simple_pipeline')
>>> runner.run(is_even_pipeline, 3)
isoddop called with [4]
False
```

after_execution(*op: chariots.base._base_op.BaseOp, args: List[Any], output: Any*)

called after the operation has been executed (and after it's *after_execution*'s method).

Parameters

- **op** – the operation that was executed
- **args** – the arguments that were passed to the op

- **output** – the output the op produced. DO NOT MODIFY the output reference as it might cause some undefined behavior

before_execution(*op: chariots.base._base_op.BaseOp, args: List[Any]*)
called before the operation is executed (and before the operation's *before_execution*'s method).

Parameters

- **op** – the operation that is going to be executed
- **args** – the list of arguments that are going to be passed to the operation. DO NOT MODIFY those references as this might cause some undefined behavior

class chariots.callbacks.**PipelineCallback**
Bases: object

a pipeline callback is used to define instructions that need to be executed at certain points in the pipeline execution:

- before the pipeline is ran
- before each node of the pipeline
- after each node of the pipeline
- after the pipeline is ran

to create your own, you need to overide one or more of the *before_execution*, *after_execution*, *before_node_execution*, *after_node_execution* methods:

```
>>> class MyPipelineLogger(PipelineCallback):
...
...     def before_execution(self, pipeline: "chariots.Pipeline", args:␣
↪List[Any]):
...         print('running {}'.format(pipeline))
...
...     def before_node_execution(self, pipeline: "chariots.Pipeline", node:
↪"BaseNode", args: List[Any]):
...         print('running {} for {}'.format(node.name, pipeline.name))
```

```
>>> is_even_pipeline = Pipeline([
...     Node(AddOneOp(), input_nodes=['__pipeline_input__'], output_nodes=
↪'modified'),
...     Node(IsOddOp(), input_nodes=['modified'],
...          output_nodes=['__pipeline_output__'])
... ], 'simple_pipeline', pipeline_callbacks=[MyPipelineLogger()])
>>> runner.run(is_even_pipeline, 3)
running <OP simple_pipeline>
running addoneop for simple_pipeline
running isoddop for simple_pipeline
False
```

after_execution(*pipeline: chariots._pipeline.Pipeline, args: List[Any], output: Any*)
called after all the nodes of the pipeline have been ran with the pipeline being run and the output of the run

Parameters

- **pipeline** – the pipeline being run
- **args** – the pipeline input that as given at the beginning of the run
- **output** – the output of the pipeline run. DO NOT MODIFY those references as this might cause some undefined behavior

after_node_execution(*pipeline: chariots._pipeline.Pipeline, node: chariots.base._base_nodes.BaseNode, args: List[Any], output: Any*)
called after each node is executed. The pipeline the node is in as well as the node are provided alongside the input/output of the node that ran

Parameters

- **pipeline** – the pipeline being run
- **node** – the node that is about to run
- **args** – the arguments that was given to the node
- **output** – the output the node produced. . DO NOT MODIFY those references as this might cause some undefined behavior

before_execution(*pipeline: chariots._pipeline.Pipeline, args: List[Any]*)
called before any node in the pipeline is ran. provides the pipeline that is being run and the pipeline input

Parameters

- **pipeline** – the pipeline being ran
- **args** – the pipeline inputs. DO NOT MODIFY those references as this might cause some undefined behavior

before_node_execution(*pipeline: chariots._pipeline.Pipeline, node: chariots.base._base_nodes.BaseNode, args: List[Any]*)
called before each node is executed the pipeline the node is in as well as the node are provided alongside the arguments the node is going to be given

Parameters

- **pipeline** – the pipeline being run
- **node** – the node that is about to run
- **args** – the arguments that are going to be given to the node. DO NOT MODIFY those references as this might cause some undefined behavior

4.3 chariots.nodes

A node represents a step in a Pipeline. It is linked to one or several inputs and can produce one or several ouptuts:

```
>>> train_logistics = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.FIT_PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')
```

you can also link the first and/or the last node of your pipeline to the pipeline input and output:

```
>>> pred = Pipeline([
...     Node(IrisFullDataSet(),input_nodes=['__pipeline_input__'], output_nodes=["x
↪"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_nodes=[
↪'__pipeline_output__'])
... ], 'pred')
```

class chariots.nodes.**Node**(*op: chariots.base._base_op.BaseOp, input_nodes: Optional[List[Union[str, chariots.base._base_nodes.BaseNode]]] = None, output_nodes: Union[List[Union[str, chariots.base._base_nodes.BaseNode]], str, chariots.base._base_nodes.BaseNode] = None*)

Bases: chariots.base._base_nodes.BaseNode

Class that handles the interaction between a pipeline and an Op. it handles defining the nodes that are going to be used as the inputs of the op and how the output of the op should be reppresented for the rest of the pipeline.

```
>>> train_logistics = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.FIT_PREDICT), input_nodes=["x"], output_nodes="x_
↪transformed"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')
```

you can also link the first and/or the last node of your pipeline to the pipeline input and output:

```
>>> pred = Pipeline([
...     Node(IrisFullDataSet(),input_nodes=['__pipeline_input__'], output_nodes=[
↪"x"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes=['__pipeline_output__'])
... ], 'pred')
```

Parameters

- **op** – the op this Node represents
- **input_nodes** – the input_nodes that are going to be used as inputs of the inner op the node, the inputs will be given to the op in the order they are defined in this argument.
- **output_nodes** – a symbolic name for the the output(s) of the op, if the op returns a tuple *output_noes* should be the same length as said tuple

check_version_compatibility(*upstream_node: chariots.base._base_nodes.BaseNode, store_to_look_in: chariots._op_store.OpStore*)

checks that this node is compatible with a potentially new version of an upstream node‘

Parameters

- **upstream_node** – the upstream node to check for version compatibality with
- **store_to_look_in** – the op_store to look for valid relationships between this node and upstream versions

Raises ***VersionError*** – when the two nodes are not compatible

execute(*params: List[Any], runner: Optional[chariots.base._base_runner.BaseRunner] = None*) → Any

executes the underlying op on params

Parameters

- **runner** – runner that can be provided if the node needs one (mostly if node is a pipeline)
- **params** – the inputs of the underlying op

Raises **ValueError** – if the runner is not provided but needed

Returns the output of the op

property is_loadable

Returns whether or not this node and its inner op can be loaded

load_latest_version(*store_to_look_in: chariots._op_store.OpStore*) → Optional[chariots.base._base_nodes.BaseNode]

reloads the latest version of the op this node represents by looking for available versions in the store

Parameters **store_to_look_in** – the store to look for new versions in

Returns the reloaded node if any older versions where found in the store otherwise *None*

property name

the name of the node. by default this will be the name of the underlying op.

property node_version

the version of this node

persist(*store: chariots._op_store.OpStore, downstream_nodes: Optional[List[BaseNode]]*) → Optional[chariots.versioning._version.Version]

persists this nodes's data (usually this means saving the serialized bytes of the inner op of this node (for the *Node* class

Parameters

- **store** – the store in which to store the node
- **downstream_nodes** – the node(s) that are going to accept the current version of this node as upstream

property requires_runner

whether or not this node requires a runner to be executed (typically if the inner op is a pipelines)

class chariots.nodes.**DataLoadingNode**(*serializer: chariots.base._base_serializer.BaseSerializer, path: str, output_nodes=None, name: Optional[str] = None, saver: Optional[chariots.base._base_saver.BaseSaver] = None*)

Bases: chariots.nodes._data_node.DataNode

a node for loading data from the ap's saver (if used in an app, otherwise use the *attach_save* method to define this node's saver).

You can use this node like any other node except that it doesn't take a *input_nodes* parameters

```
>>> load_and_analyse_iris = Pipeline([
...     DataLoadingNode(serializer=CSVSerializer(), path='/train.csv', output_
↪nodes=["train_df"]),
...     Node(AnalyseDataSetOp(), input_nodes=["train_df"], output_nodes=["__
↪pipeline_output__"]),
... ], "analyse")
```

then you can prepare the pipeline (which attaches the saver) and run the pipeline

```
>>> load_and_analyse_iris.prepare(saver)
>>> runner.run(load_and_analyse_iris)
Counter({1: 39, 2: 38, 0: 35})
```

Parameters

- **saver** – the saver to use for loading or saving data (if not specified at init, you can use the *attach_saver* method

- **serializer** – the serializer to use to load the dat
- **path** – the path to load the data from
- **output_nodes** – an optional symbolic name for the node to be called by other node. If this node is the output of the pipeline use "pipeline_output" or *ReservedNodes.pipeline_output*
- **name** – the name of the op

execute(*params: List[Any], runner: Optional[chariots.base._base_runner.BaseRunner] = None*) → Any

executes the computation represented byt this node (loads/saves dataset for dataset nodes, executes underlyin op for *Node*

Parameters **params** – the inputs provided by the *input_nodes*

Returns the output(s) of the node

property node_version

the version of this node

class chariots.nodes.**DataSavingNode**(*serializer: chariots.base._base_serializer.BaseSerializer, path: str, input_nodes: Optional[List[Union[AnyStr, Node]]], name: Optional[str] = None, saver: Optional[chariots.base._base_saver.BaseSaver] = None*)

Bases: chariots.nodes._data_node.DataNode

a node for saving data into the app's Saver (if used in an app, otherwise use the *attach_save* method to define this node's saver).

You can use this node like any other node except that it doesn't take a *input_nodes* parameters

```
>>> save_train_test = Pipeline([
...     Node(IrisDF(), output_nodes='df'),
...     Node(TrainTestSplit(), input_nodes=['df'], output_nodes=['train_df',
↪'test_df']),
...     DataSavingNode(serializer=CSVSerializer(), path='/train.csv', input_
↪nodes=['train_df']),
...     DataSavingNode(serializer=DillSerializer(), path='/test.pkl', input_
↪nodes=['test_df'])
... ], "save")
```

you can then use the prepare method of the pipeline to attach a saver to our various *DataNodes* and run the pipeline like any other

```
>>> save_train_test.prepare(saver)
>>> runner.run(save_train_test)
```

Parameters

- **saver** – the saver to use for loading or saving data (if not specified at init, you can use the *attach_saver* method
- **serializer** – the serializer to use to load the dat
- **path** – the path to load the data from
- **input_nodes** – the data that needs to be saved
- **name** – the name of the op

execute(*params: List[Any], runner: Optional[chariots.base._base_runner.BaseRunner] = None*) → Any
executes the computation represented byt this node (loads/saves dataset for dataset nodes, executes underlyin op for *Node*

Parameters **params** – the inputs provided by the *input_nodes*

Returns the output(s) of the node

property node_version
the version of this node

class chariots.nodes.**ReservedNodes**
Bases: enum.Enum

enum of reserved node names

pipeline_input = '__pipeline_input__'

pipeline_output = '__pipeline_output__'

property reference
the output references of the reserved nodes

4.4 chariots.ops

operations are the atomic computation element of Chariots, you can use them to train models, preprocess your data, extract features and much more.

to create your own operations, you will need to subclass one of the base op classes:

- create a minimalist operation by subclassing the *BaseOp class*
- create an op that supports loading and saving by subclassing the *LoadableOp* class
- create a machine learning operation by subclassing on of the machine learning ops (depending on your framework) like an *sklearn op*

class chariots.ops.**LoadableOp**(*op_callbacks: Optional[List[chariots.callbacks._op_callback.OpCallBack]] = None*)
Bases: chariots.base._base_op.BaseOp

an operation that supports loading and saving. This means that when a pipeline tries to load a node using this kind of op, it will try to find the serialized bytes of the last saved version of this op and pass them to the *load* method of the op.

Similarly when the pipeline will try to save a node using this kind of operation, it will get the op's serialized bytes by calling it's *serialize* method (along with the op's version)

to create your own loadable op, you will need to: - define the *load* and *serialize* method - define the *execute* method as for a normal op to define the behavior of your op

execute(**args*, ***kwargs*)
main method to override. it defines the behavior of the op. In the pipeline the argument of the pipeline will be passed from the node with one argument per input (in the order of the input nodes)

load(*serialized_object: bytes*)
Receives serialize bytes of a newer version of this class and sets the internals of he op accordingly.

Parameters **serialized_object** – the serialized bytes of this op (as where outputed by the *serialize* method

serialize() → bytes
serializes the object into bytes (to be persisted with a Saver) to be reloaded in the future (you must ensure the compatibility with the *load* method

Returns the serialized bytes representing this operation

4.5 chariots.runners

runners are used to execute Pipelines: they define in what order and how each node of the pipeline should be executed.

For the moment Chariots only provides a basic sequential runner that executes each operation of a pipeline one after the other in a single threat however we have plans to introduce new runners (process and thread based ones as well as some cluster computing one) in future releases.

You can use runners directly if you want to execute your pipeline manually:

```
>>> runner = SequentialRunner()
>>> runner.run(is_odd_pipeline, 5)
True
```

or you can set the default runner of your app and it will be used every time a pipeline execution is called:

```
>>> my_app = Chariots(app_pipelines=[is_odd_pipeline], runner=SequentialRunner(),␣
↪path=app_path,
...                    import_name="my_app")
```

class chariots.runners.**SequentialRunner**
Bases: chariots.base._base_runner.BaseRunner

runner that executes every node in a pipeline sequentially in a single thread.

run(*pipeline: chariots._pipeline.Pipeline*, *pipeline_input: Optional[Any] = None*)
runs a pipeline, provides it with the correct input and extracts the results if any

Parameters

- **pipeline** – the pipeline to run
- **pipeline_input** – the input to be given to the pipeline

Returns the output of the graph called on the input if applicable

4.6 chariots.savers

savers are used to persist and retrieve information about ops, nodes and pipeline (such as versions, persisted versions, datasets, and so on).

A saver can be viewed as the basic abstraction of a file system (interprets path) and always has a root path (that represents the path after which the saver will start persisting data).

For now chariots only provides a basic *FileSaver* saver but there are plans to add more in future releases (in particular to support bottomless cloud storage solutions such as aws s3 and Google cloud storage).

to create your own saver, you can subclass the *BaseSaver class*

To use a specific saver in your app, you will need to specify the saver class and the root path of the saver in the *Chariots* initialisation:

```
>>> my_app = Chariots(app_pipelines=my_pipelines, path=app_path, saver_cls=FileSaver,␣
↪import_name="my_app")
```

class chariots.savers.**FileSaver**(*root_path: str*)
Bases: chariots.base._base_saver.BaseSaver

a saver that persists to the local file system of the machine the *Chariots* saver is running on.

load(*path: str*) → bytes
loads the bytes serialized at a specific path

Parameters **path** – the path to load the bytes from.You should not include the *root_path* of the saver in this path: loading to */foo/bar.txt* on a saver with */my/root/path* as root path will load */my/root/path/foo/bar.txt*

Returns saved bytes

save(*serialized_object: bytes*, *path: str*) → bool
saves bytes to a specific path.

Parameters

- **serialized_object** – the bytes to persist
- **path** – the path to save the bytes to. You should not include the *root_path* of the saver in this path: saving to */foo/bar.txt* on a saver with */my/root/path* as root path will create/update */my/root/path/foo/bar.txt*

Returns whether or not the object was correctly serialized.

4.7 chariots.serializers

Serializers are utils classes that are used throughout the *Chariots* framework to transform objects into bytes. there are for instance used to serialize the inner models of the machine learning ops:

```
>>> class LinearRegression(SKSupervisedOp):
...
...     serializer_cls = MySerializerCls
...
...     model_class = PCA
```

there are also usually used in the saving nodes to choose the serialization method for your datasets:

```
>>> saving_node = DataSavingNode(serializer=CSVSerializer(), path='my_path.csv',␣
↪input_nodes=["my_dataset"])
```

class chariots.serializers.**DillSerializer**
Bases: chariots.base._base_serializer.BaseSerializer

serializes objects using the dill library (similar to pickle but optimized for numpy arrays.

deserialize_object(*serialized_object: bytes*) → Any
returns the deserialized object from serialized bytes (that will be loaded from a saver)

Parameters **serialized_object** – the serialized bytes

Returns the deserialized objects

serialize_object(*target: Any*) → bytes
serializes the object into bytes (for ml ops *target* will be the model itself and not the op, for the data ops the *target* will be the input of the node)

Parameters **target** – the object that will be serialized

Returns the bytes of the serialized object

class chariots.serializers.**JSONSerializer**
Bases: chariots.base._base_serializer.BaseSerializer

serializes objects into JSON format

deserialize_object(*serialized_object: bytes*) → Any
returns the deserialized object from serialized bytes (that will be loaded from a saver)

Parameters **serialized_object** – the serialized bytes

Returns the deserialized objects

serialize_object(*target: Any*) → bytes
serializes the object into bytes (for ml ops *target* will be the model itself and not the op, for the data ops the *target* will be the input of the node)

Parameters **target** – the object that will be serialized

Returns the bytes of the serialized object

class chariots.serializers.**CSVSerializer**
Bases: chariots.base._base_serializer.BaseSerializer

A serializer to save a pandas data frame.

Raises **Typeerror** – if the node receives something other than a pandas *DataFrame*

deserialize_object(*serialized_object: bytes*) → pandas.core.frame.DataFrame
returns the deserialized object from serialized bytes (that will be loaded from a saver)

Parameters **serialized_object** – the serialized bytes

Returns the deserialized objects

serialize_object(*target: pandas.core.frame.DataFrame*) → bytes
serializes the object into bytes (for ml ops *target* will be the model itself and not the op, for the data ops the *target* will be the input of the node)

Parameters **target** – the object that will be serialized

Returns the bytes of the serialized object

4.8 chariots.sklearn

the sklearn module provides support for the scikit-learn framework.

this module provides two main classes (*SKSupervisedOp*, *SKUnsupervisedOp*) that need to be subclassed to be used. to do so you will need to set the *model_class* class attribute and potentially the *model_parameters* class attribute. this should be a *VersionedFieldDict* which defines the parameters your model should be initialized with. As for other machine learning ops, you can override the *training_update_version* class attribute to define which version will be changed when the operation is retrained:

```
>>> class PCAOp(SKUnsupervisedOp):
...     training_update_version = VersionType.MAJOR
...     model_parameters = VersionedFieldDict(VersionType.MAJOR, {"n_components": 2,})
...     model_class = VersionedField(PCA, VersionType.MAJOR)
```

Once your op class is define, you can use it as any MLOp choosing your *MLMode* to define the behavior of your operation (fit and/or predict):

```
>>> train_pca = Pipeline([Node(IrisXDataSet(), output_nodes=["x"]), Node(PCAOp(MLMode.
↪FIT), input_nodes=["x"])],
...                       'train_pca')
```

class chariots.sklearn.**SKSupervisedOp**(*mode: chariots._ml_mode.MLMode*, *op_callbacks: Optional[List[chariots.callbacks._op_callback.OpCallBack]] = None*)

Bases: chariots.sklearn._base_sk_op.BaseSKOp

Op base class to create supervised models using the scikit learn framework., If using the *MLMode.FIT* or *MLMode.FIT_PREDICT*, you will need to link this op to a X and a y upstream node:

```
>>> train_logistics = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')
```

and if you are using the op with the *MLMode.PREDICT* mode you will only need to link the op to an X upstream node:

```
>>> pred = Pipeline([
...     Node(IrisFullDataSet(),input_nodes=['__pipeline_input__'], output_nodes=[
↪"x"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes=['__pipeline_output__'])
... ], 'pred')
```

fit(*X*, *y*)

method used by the operation to fit the underlying model

DO NOT TRY TO OVERRIDE THIS METHOD.

Parameters

- **X** – the input that the underlying supervised model will fit on (type must be compatible with the sklearn lib such as numpy arrays or pandas data frames)
- **y** – the output that hte underlying supervised model will fit on (type must be compatible with the sklearn lib such as numpy arrays or pandas data frames)

predict(*X*) → Any

method used internally by the op to predict with the underlying model.

DO NOT TRY TO OVERRIDE THIS METHOD.

Parameters **X** – the input the model has to predict on. (type must be compatible with the sklearn lib such as numpy arrays or pandas data frames)

class chariots.sklearn.**SKUnsupervisedOp**(*mode: chariots._ml_mode.MLMode*, *op_callbacks: Optional[List[chariots.callbacks._op_callback.OpCallBack]] = None*)

Bases: chariots.sklearn._base_sk_op.BaseSKOp

base class to create unsupervised models using the scikit-learn framework. Whatever the mode you will need to link this op with a single upstream node:

```
>>> train_logistics = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')

>>> pred = Pipeline([
...     Node(IrisFullDataSet(),input_nodes=['__pipeline_input__'], output_nodes=[
↪"x"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes=['__pipeline_output__'])
... ], 'pred')
```

fit(*X*)

method used to fit the underlying unsupervised model.

DO NOT TRY TO OVERRIDE THIS METHOD.

Parameters **X** – the dataset (compatible type with the sklearn lib as pandas data-frames or numpy arrays).

predict(*X*) → Any

transforms the dataset using the underlying unsupervised model

DO NOT TRY TO OVERRIDE THIS METHOD.

Parameters **X** – the dataset to transform (type must be compatible with the sklearn library such as pandas data frames or numpy arrays).

4.9 chariots.keras

class chariots.keras.**KerasOp**(*mode: chariots._ml_mode.MLMode*, *verbose: Optional[int] = 1*)

Bases: chariots.base._base_ml_op.BaseMLOp

Keras Ops help you create ops for all your Keras based neural networks.

To create your keras op, you will need to:

- define the initialisation behavior of your model by overriding the *_init_model* method.
- define any additional training parameters using the *fit_params VersionedFieldDict*.

```
>>> from chariots import Pipeline, MLMode
>>> from chariots.keras import KerasOp
>>> from chariots.nodes import Node
>>> from chariots.versioning import VersionType, VersionedFieldDict
>>> from keras import models, layers
```

(continues on next page)

(continued from previous page)

```
...
...
>>> class KerasLinear(KerasOp):
...     fit_params = VersionedFieldDict(VersionType.MAJOR, {
...         'epochs': 3,
...         'batch_size': 32,
...     })
...
...     def _init_model(self, *input_data_sets):
...         model = models.Sequential([layers.Dense(3, activation='softmax',␣
↪input_shape=(4,))])
...         model.compile(loss='categorical_crossentropy', optimizer='adam')
...         return model
...
...
>>> train = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["X", "y"]),
...     Node(Categorize(), input_nodes=['y'], output_nodes='y_cat'),
...     Node(KerasLinear(mode=MLMode.FIT, verbose=0), input_nodes=['X', 'y_cat'])
... ], 'train')
>>> pred = Pipeline([
...     Node(KerasLinear(mode=MLMode.PREDICT), input_nodes=['__pipeline_input__'],
...          output_nodes='__pipeline_output__')
... ], 'pred')
```

than you can call your pipeline as you would with any other:

```
>>> runner.run(train)
...
>>> runner.run(pred, np.array([[1, 2, 3, 4]]))
array([[...]], dtype=float32)
```

or use them in an app:

```
>>> app = Chariots([train, pred], app_path, import_name='my_app')
```

fit(*input_data_sets: Union[List[numpy.ndarray], numpy.ndarray], output_datasets: Union[List[numpy.ndarray], numpy.ndarray]*)
fits the inner model of the op on data (in args and kwargs) this method must not return any data (use the FIT_PREDICT mode to predict on the same data the op was trained on)

input_params = <chariots.versioning._versioned_field_dict.VersionedFieldDict object>

predict(*input_datasets*) → Any
the method used to do predictions/inference once the model has been fitted/loaded

4.10 chariots.versioning

The versioning module provides all the types the Chariot's versioning logic is built around. If you want to know more about the way semantic versioning is handled in Chariots, you can go check out the *guiding principles*.

This module is built around the *VersionableMeta* metaclass. This is a very simple metaclas that adds the __version__ class attribute whenever a new versionable class is created:

```
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     pass
>>> MyVersionedClass.__version__
<Version, major:da39a, minor: da39a, patch: da39a>
```

to control the version of your class, you can use *VersionedField* descriptors:

..doctest:

```
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     foo = VersionedField(3, VersionType.MINOR)
>>> MyVersionedClass.__version__
<Version, major:94e72, minor: 36d3c, patch: 94e72>
>>> MyVersionedClass.foo
3
```

and if in a future version of your code, the class attribute changes, the subsequent version will be changed:

..doctest:

```
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     foo = VersionedField(5, VersionType.MINOR)
>>> MyVersionedClass.__version__
<Version, major:94e72, minor: 72101, patch: 94e72>
>>> MyVersionedClass.foo
5
```

but this version change only happen when the class is created and not when you change the value of this class attribute during the lifetime of your class:

```
>>> MyVersionedClass.foo = 7
>>> MyVersionedClass.__version__
<Version, major:94e72, minor: 72101, patch: 94e72>
>>> MyVersionedClass.foo
7
```

This module also provides a helper for creating versioned *dict* (where each value of the *dict* acts as a *VersionedField*) with the *VersionedFieldDict* descriptors:

```
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     versioned_dict = VersionedFieldDict(VersionType.PATCH,{
...         'foo': 1,
...         'bar': 2,
...         'blu': VersionedField(3, VersionType.MAJOR)
...     })
>>> MyVersionedClass.__version__
<Version, major:ddf7a, minor: 1b365, patch: 68722>
>>> MyVersionedClass.versioned_dict['foo']
1
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     versioned_dict = VersionedFieldDict(VersionType.PATCH,{
...         'foo': 10,
...         'bar': 2,
...         'blu': VersionedField(3, VersionType.MAJOR)
...     })
>>> MyVersionedClass.__version__
<Version, major:ddf7a, minor: 1b365, patch: 18615>
>>> MyVersionedClass.versioned_dict['foo']
```

(continues on next page)

(continued from previous page)

```
10
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     versioned_dict = VersionedFieldDict(VersionType.PATCH,{
...         'foo': 1,
...         'bar': 2,
...         'blu': VersionedField(10, VersionType.MAJOR)
...     })
>>> MyVersionedClass.__version__
<Version, major:d5abf, minor: 1b365, patch: 68722>
>>> MyVersionedClass.versioned_dict['blu']
10
```

this is for instance used for the *model_parameters* attribute of the *sci-kit learn ops*

class chariots.versioning.**Version**(*major: Union[_hashlib.HASH, str, None] = None*, *minor: Union[_hashlib.HASH, str, None] = None*, *patch: Union[_hashlib.HASH, str, None] = None*, *creation_time: Optional[float] = None*)

Bases: `object`

Type of all the different versions used throughout the Chariots framework.

A Chariots version has three subversions (major, minor, patch) each subversion is the hexadecimal representation of the VersionedFields of this version.

two versions are considered equal if all their subversions are the same. A version is considered greater than the other of the other if one or more of it's subversions is different and it has been created later.

you can use the + operation between two version to create a new version. this new version will NOT be the same as creating the new version from the same VersionedFields as the two versions: *version(foo) + version(bar) != version(foo, bar)*

property creation_time
the time stamp of the creation time of the version

property major
the hash of the major subversion

property minor
the hash of the minor subversion

classmethod parse(*version_string: str*) → chariots.versioning._version.Version
parses a string representation of a saved version and returns a valid *Version* object

Parameters **version_string** – the version string to parse (this must come from *str(my_version)* and not *repr(my_version)*

Returns the version represented by the version string

property patch
the hash of the patch subversion

update(*version_type: chariots.versioning._version_type.VersionType*, *input_bytes: bytes*) → chariots.versioning._version.Version
updates the corresponding subversion of this version with some bytes

Parameters

- **version_type** – the subversion to update
- **input_bytes** – the bytes to update the subversion with

Returns the updated version

update_major(*input_bytes: bytes*) → chariots.versioning._version.Version
updates the major subversion with some bytes

Parameters **input_bytes** – bytes to update the major subversion with

Returns the updated version

update_minor(*input_bytes: bytes*) → chariots.versioning._version.Version
updates the minor subversion with some bytes

Parameters **input_bytes** – bytes to update the minor subversion with

Returns the updated version

update_patch(*input_bytes: bytes*) → chariots.versioning._version.Version
updates the patch subversion with some bytes

Parameters **input_bytes** – bytes to update the patch subversion with

Returns the updated version

class chariots.versioning.**VersionType**
Bases: enum.Enum

am enum to give the three subversion types used in the chariots framework

MAJOR = 'major'

MINOR = 'minor'

PATCH = 'patch'

class chariots.versioning.**VersionedField**(*value: Any*, *affected_version: chariots.versioning._version_type.VersionType*)
Bases: object

a descriptor to mark that a certain class attribute has to be incorporated in a subversion a versioned field is used as a normal class attribute (when gotten it returns the inner value) but is used to generate the version of the class it is used on when said class is created (at import time)

```
>>> class MyVersionedClass(metaclass=VersionableMeta):
...     foo = VersionedField(3, VersionType.MINOR)
>>> MyVersionedClass.foo
3
```

Parameters

- **value** – the inner value to be given the field whcih will be returned when you try to get the class attribute
- **affected_version** – the subversion this class attribute has to affect

class chariots.versioning.**VersionedFieldDict**(*default_version=<VersionType.MAJOR: 'major'>*, **args*, ***kwargs*)
Bases: collections.abc.MutableMapping

a versioned field dict acts as a normal dictionary but the values as interpreted as versioned fields when it is a VersionedClass class attribute

property version_dict
property to retrieve the name of the fields and the Versions associated to each of them :return: the mapping with the key and the version of the value

class chariots.versioning.**VersionableMeta**(*clsname*, *superclasses*, *attributedict*)
Bases: type

metaclass for all versioned objects in the library. When a new class using this metaclas is created, it will have a *__version__* class attribute that sets all the subversions of the class depending on the VersionedFields the class was created with

4.11 chariots.cli

Console script for chariots.

4.12 chariots.errors

exception chariots.errors.**BackendError**
Bases: ImportError

exception chariots.errors.**VersionError**
Bases: TypeError

static handle()

class chariots.**Pipeline**(*pipeline_nodes: List[base.BaseNode]*, *name: str*, *pipeline_callbacks: Optional[List[chariots.callbacks._pipeline_callback.PipelineCallback]] = None*)
a pipeline is a collection of linked nodes that have to be executed one on top of each other. pipelines are the main way to use Chariots.

to build a simple pipeline you can do as such:

```
>>> pipeline = Pipeline([
...     Node(AddOneOp(), input_nodes=["__pipeline_input__"], output_nodes=["added_
↪number"]),
...     Node(IsOddOp(), input_nodes=["added_number"], output_nodes=["__pipeline_
↪output__"])
... ], "simple_pipeline")
```

here we have just created a very simple pipeline with two nodes, one that adds one to the provided number and one that returns whether or not the resulting number is odd

to use our pipeline, we can either do it manually with a runner:

```
>>> from chariots.runners import SequentialRunner
>>> runner = SequentialRunner()
>>> runner.run(pipeline=pipeline, pipeline_input=4)
True
```

you can also as easily deploy your pipeline to a Chariots app (small micro-service to run your pipeline)

```
>>> from chariots import Chariots
>>> app = Chariots([pipeline], path=app_path, import_name="simple_app")
```

Once this is done you can deploy your app as a flask app and get the result of the pipeline using a client:

```
>>> client.call_pipeline(pipeline, 4)
True
```

Parameters

- **pipeline_nodes** – the nodes of the pipeline. each node has to be linked to previous node (or *__pipeline_input__*). nodes can create branches but the only output remaining has to be *__pipeline_output__* (or no ouptut)
- **name** – the name of the pipeline. this will be used to create the route at which to query the pipeline in the Chariots app
- **pipeline_callbacks** – callbacks to be used with this pipeline (monitoring and logging for instance)

execute(*runner: chariots.base._base_runner.BaseRunner*, *pipeline_input=None*)
present for inheritance purposes from the Op Class, this will automatically raise

execute_node(*node: chariots.base._base_nodes.BaseNode, intermediate_results: Dict[NodeReference, Any], runner: chariots.base._base_runner.BaseRunner*)
executes a node from the pipeline, this method is called by the runners to make the pipeline execute one of it's node and all necessary callbacks

Parameters

- **node** – the node to be executed
- **intermediate_results** – the intermediate result to look in in order to fin the node's inputs
- **runner** – a runner to be used in case the node needs a runner to be executed (internal pipeline)

Raises **ValueError** – if the output of the node does not correspond to the length of it's output references

Returns the final result of the node after the execution

static extract_results(*results: Dict[chariots.base._base_nodes.NodeReference, Any]*) → Any
extracts the output of a pipeline once all the nodes have been computed. This method is used by runners when once all the nodes are computed in order to check and get the final result to return

Parameters **results** – the outputs left unused once the graph has been ran.

Raises **ValueError** – if some output was unused once every node is computed and the remaining is not the output of the pipeline

Returns the final result of the pipeline as needs to be returned to the use

get_pipeline_versions() → Mapping[chariots.base._base_nodes.BaseNode, chariots.versioning._version.Version]
returns the versions of every op in the pipeline

Returns the mapping version for node

load(*op_store: chariots._op_store.OpStore*)
loads all the latest versions of the nodes in the pipeline if they are compatible from an *OpStore*. if the latest version is not compatible, it will raise a *VersionError*

Parameters **op_store** – the op store to look for existing versions if any and to load the bytes of said version if possible

Raises ***VersionError*** – if a node is incompatible with one of it's input. For instance if a node has not been trained on the latest version of it's input in an inference pipeline

Returns this pipeline once it has been fully loaded

property name
the name of the pipeline

property node_for_name
utils mapping that has node names in input and the nodes objects in values

property nodes
the nodes of the pipeline

prepare(*saver: chariots.base._base_saver.BaseSaver*)
prepares the pipeline to be served. This is manly used to attach the correct saver to the nodes that need one (data saving and loading nodes for instance).

Parameters **saver** – the saver to attach to all the nodes that need one

save(*op_store: chariots._op_store.OpStore*)
persists all the nodes (that need saving) in an *OpStore*. this is used for instance when a training pipeline has been executed and needs to save it's trained node(s) for the inference pipeline to load them. This method also updates the versions available for the store to serve in the future

Parameters **op_store** – the store to persist the nodes and their versions in

class chariots.**Chariots**(*app_pipelines: List[chariots._pipeline.Pipeline], path, saver_cls: Type[chariots.base._base_saver.BaseSaver] = <class 'chariots.savers._file_saver.FileSaver'>, runner: Optional[chariots.base._base_runner.BaseRunner] = None, default_pipeline_callbacks: Optional[List[chariots.callbacks._pipeline_callback.PipelineCallback]] = None, *args, **kwargs*)
small *Flask* application used to rapidly deploy pipelines:

```
>>> my_app = Chariots(app_pipelines=[is_odd_pipeline], path=app_path, import_name=
↪"my_app")
```

you can then deploy the app as you would with the flask comand:

```
$ flask
```

or if you have used *the chariots' template*, you can use the predefined cli once the project is installed:

```
$ my_great_project start
```

once the app is started you can use it with the client (that handles creating the requests and serializing to the right format) to query your pipelines:

```
>>> client.call_pipeline(is_odd_pipeline, 4)
False
```

alternatively, you can query the *Chariots* server directly as you would for any normal micro-service. The server has the following routes:

- */pipelines/<pipeline_name>/main*
- */pipelines/<pipeline_name>/versions*
- */pipelines/<pipeline_name>/load*
- */pipelines/<pipeline_name>/save*
- */pipelines/<pipeline_name>/health_check*

for each pipeline that was registered to the *Chariots* app. It also creates some common routes for all pipelines:

- */health_check*
- */available_pipelines*

Parameters

- **app_pipelines** – the pipelines this app will serve
- **path** – the path to mount the app on (whether on local or remote saver). for isntance using a *LocalFileSaver* and '/chariots' will mean all the information persisted by the *Chariots* server (past versions, trained models, datasets) will be persisted there
- **saver_cls** – the saver class to use. if None the *FileSaver* class will be used as default
- **runner** – the runner to use to run the pipelines. If None the *SequentialRunner* will be used as default
- **default_pipeline_callbacks** – pipeline callbacks to be added to every pipeline this app will serve.
- **args** – additional positional arguments to be passed to the Flask app
- **kwargs** – additional keywords arguments to be added to the Flask app

class chariots.**Client**(*backend_url: str = 'http://127.0.0.1:5000'*)

Client to query/save/load the pipelines served by a (remote) *Chariots* app.

for instance if you have built your app as such and deployed it:

```
>>> train_pca = Pipeline([Node(IrisXDataSet(), output_nodes=["x"]),␣
↪Node(PCAOp(mode=MLMode.FIT),
...                         input_nodes=["x"])], "train_pca")

>>> train_logistic = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
... ], 'train_logistics')

>>> pred = Pipeline([
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["__pipeline_input__"], output_
↪nodes="x_transformed"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_
↪nodes="__pipeline_output__")
... ], "pred")

>>> app = Chariots([train_pca, train_logistic, pred], app_path, import_name="iris_
↪app")
```

you can then train save and load your pipelines remotely from the client

```
>>> client.call_pipeline(train_pca)
>>> client.save_pipeline(train_pca)
>>> client.load_pipeline(train_logistic)
>>> client.call_pipeline(train_logistic)
>>> client.save_pipeline(train_logistic)
>>> client.load_pipeline(pred)
>>> client.call_pipeline(pred, [[1, 2, 3, 4]])
[1]
```

but if you execute them in the wrong order the client will propagate the errors that occur on the *Chariots* server

```
>>> client.call_pipeline(train_pca)
>>> client.save_pipeline(train_pca)
>>> client.load_pipeline(pred)
Traceback (most recent call last):
...
chariots.errors.VersionError: the pipeline you requested cannot be loaded because␣
↪of version incompatibilityHINT: retrain and save/reload in order to have a␣
↪loadable version
```

this example is overkill as you can use *MLMode.FitPredict* flag (not used here to demonstrate the situations where *VersionError* will be raised). this would reduce the amount of saving/loading to get to the prediction.

call_pipeline(*pipeline: chariots._pipeline.Pipeline*, *pipeline_input: Optional[Any] = None*) → Any

sends a request to the *Chariots* server in order to get this pipeline executed remotely on the server.

```
>>> client.call_pipeline(is_odd_pipeline, 4)
False
>>> client.call_pipeline(is_odd_pipeline, 5)
True
```

here you can get the user gets the output of the pipeline that got executed in our *Chariots* micro service

Parameters

- **pipeline** – the pipeline that needs to be executed in the remote *Chariots* server
- **pipeline_input** – the input of the pipeline (will be provided to the node with *__pipeline__input__* in it's *input_nodes*). If none of the nodes accept a __pipeline_input__ and this is provided the execution of the pipeline will fail. pipeline_input needs to be JSON serializable

Raises

- **ValueError** – if the pipeline requested is not present in the *Chariots* app.
- **ValueError** – if the execution of the pipeline fails

Returns the result of the pipeline. it needs to be JSON serializable for chariots to be able to pass it through http

is_pipeline_loaded(*pipeline: chariots._pipeline.Pipeline*) → bool

checks whether or not the pipeline has been loaded

Parameters **pipeline** – the pipeline to check

load_pipeline(*pipeline: chariots._pipeline.Pipeline*)

reloads all the nodes in a pipeline. this is usually used to load the updates of a node/model in the inference pipeline after the training pipeline(s) have been executed. If the latest version of a saved node is incompatible with the rest of the pipeline, this will raise a *VersionError*

Parameters **pipeline** – the pipeline to reload

Raises ***VersionError*** – If there is a version incompatibility between one of the nodes in the pipeline and one of it's inputs

pipeline_versions(*pipeline: chariots._pipeline.Pipeline*) → Mapping[str, chariots.versioning._version.Version]

gets all the versions of the nodes of the pipeline (different from *pipeline.get_pipeline_versions* as the client will return the version of the loaded/trained version on the (remote) *Chariots* server)

Parameters **pipeline** – the pipeline to get the versions for

Returns mapping with the node names in keys and the version object in value

save_pipeline(*pipeline: chariots._pipeline.Pipeline*)
persists the state of the pipeline on the remote *Chariots* server (usually used for saving the nodes that were trained in a train pipeline in order to load them inside the inference pipelines).

Parameters **pipeline** – the pipeline to save on the remote server. Beware: any changes made to the *pipeline* param will not be persisted (Only changes made on the remote version of the pipeline)

class chariots.**TestClient**(*app: chariots._deployment.app.Chariots*)
mock up of the client to test a full app without having to create a server

class chariots.**OpStore**(*saver: chariots.base._base_saver.BaseSaver*)
Bases: object

A Chariots OpStore handles the persisting of Ops and their versions as well as the accepted versions of each op's inputs.

the OpStore persists all this metadata about persisted ops in the */_meta.json* file using the saver provided at init

all the serialized ops are saved at /models/<op name>/<version>

The OpStore is mostly used by the Pipelines and the nodes at saving time to:

- persist the ops that they have updated
- register new versions
- register links between different ops and different versions that are valid (for instance this versions of the PCA is valid for this new version of the RandomForest

and at loading time to:

- check latest available version of an op
- check if this version is valid with the rest of the pipeline
- recover the bytes of the latest version if it is valid

the OpStore identifies op's by there name (usually a snake case of the Class of your op) so changing this name (or changing the class name) might make it hard to recover the metadata and serialized bytes of the Ops

param saver the saver the op_store will use to retrieve it's metadata and subsequent ops

get_all_versions_of_op(*op: chariots.base._base_op.BaseOp*) → Optional[List[chariots.versioning._version.Version]]
returns all the available versions of an op ever persisted in the OpGraph (or any Opgraph using the same _meta.json)

Parameters **op** – the op to get the previous persisted versions

get_op_bytes_for_version(*op: chariots.base._base_op.BaseOp*, *version: chariots.versioning._version.Version*) → bytes
loads the persisted bytes of op for a specific version

Parameters

- **op** – the op that needs to be loaded
- **version** – the version of the op to load

Returns the bytes of the op

get_validated_links(*downstream_op_name: str*, *upstream_op_name: str*) → Optional[Set[chariots.versioning._version.Version]]

register_valid_link(*downstream_op: Optional[str], upstream_op: str, upstream_op_version: chariots.versioning._version.Version*)

registers a link between an upstream and a downstream op. This means that in future relaods the downstream op will whitelist this version for this upstream op

Parameters

- **downstream_op** – the op that needs to whitelist one of it's inputs' new version
- **upstream_op** – the op that is getting whitelisted as one of the inputs of the downstream op
- **upstream_op_version** – the valid version of the op that is getting whitelisted

Returns

save()

persists all the metadata about ops and versions available in the store using the store's saver.

The saved metadata can be found at */_meta.json* from the saver's route.

save_op_bytes(*op_to_save: chariots.base._base_op.BaseOp*, *version: chariots.versioning._version.Version*, *op_bytes: bytes*)

saves op_bytes of a specific op to the path /models/<op name>/<version>.

the version that is used here is the node version (and not the op_version) as nodes might be able to modify some behaviors of the versioning of their underlying op

Parameters

- **op_to_save** – the op that needs to be saved (this will not be saved as is - only the bytes)
- **version** – the exact version to be used when persisting
- **op_bytes** – the bytes of the op to save that will be persisted

class chariots.**MLMode**

Bases: enum.Enum

mode in which to put the op (prediction of training) enum

FIT = 'fit'

FIT_PREDICT = 'fit_predict'

PREDICT = 'predict'

CHAPTER

FIVE

CHARIOTS TEMPLATE

chariots provides a template to create Chariot templates that take care of the boilerplate involved. This template is inspired by the DataScience audreyr/cookiecutter-pypackage and drivendata/cookiecutter-data-science project templates so you may find some similarities

to create a new project, just use

```
$ chariots new
```

you can then follow the prompts to customize your template (if you don't know what to put, follow the defaults). If you want a minimalist example (using the classic iris dataset), you can put *y'in the 'use_iris_example* parameter.

5.1 File Structure

the file structure of the project is as follows:

```
.
├── AUTHORS.rst
├── LICENSE
├── MANIFEST.in
├── Makefile
├── README.rst
├── docs
│   ├── Makefile
│   ├── authors.rst
│   ├── conf.py
│   ├── index.rst
│   ├── installation.rst
│   ├── make.bat
│   └── modules.rst
├── iris
│   ├── __init__.py
│   ├── app.py
│   ├── cli.py
│   ├── ops
│   │   ├── __init__.py
│   │   ├── data_ops
│   │   │   └── __init__.py
│   │   ├── feature_ops
│   │   │   └── __init__.py
│   │   └── model_ops
│   │       └── __init__.py
│   └── pipelines
```

(continues on next page)

(continued from previous page)

```
│       └── __init__.py
├── iris_local
│   ├── data
│   └── ops
├── notebooks
│   └── example_notebook.ipynb
├── requirements.txt
├── requirements_dev.txt
├── setup.cfg
├── setup.py
└── tests
    └── test_server.py`
```

the *iris* folder (it will take the name of your project) is the main module of the project. It contains three main parts:

- the ops module contains all your Chariot ops. this is where most of the models/preprocessing goes (in their specific subfolders)
- the pipelines module defines the different pipelines of your project
- the app module provides the Chariots app that you can use to deploy your pipeline

the iris_local folder is where the chariots app will be mounted on (to load and save data/models) by default

the notebooks folder is where you can put you exploration and reporting notebooks

5.2 tools

the template provides several tools in order to facilitate development:

a cli interface that include

```
$ my_great_project start
```

to start the server

a makefile to build the doc, clean the project and more

and more to come... -

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/aredier/chariots/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

chariots could always use more documentation, whether as part of the official chariots docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/aredier/chariots/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *chariots* for local development.

1. Fork the *chariots* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/chariots.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv chariots
$ cd chariots/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 chariots tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/aredier/chariots/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_chariots
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

SEVEN

CREDITS

7.1 Development Lead

- Antoine Redier <antoine.redier2@gmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER

EIGHT

HISTORY

8.1 0.1.0 (2019-06-15)

- First release on PyPI.

8.2 0.2.0 (2019-06-15)

- sci-kit learn and keras integration
- multiple outputs per nodes
- project template
- tutorials

CHAPTER

NINE

CHARIOTS

chariots aims to be a complete framework to build and deploy versioned machine learning pipelines.

- Documentation: https://chariots.readthedocs.io.

9.1 Getting Started: 30 seconds to Chariots:

You can check the documentation for a complete tutorial on getting started with chariots, but here are the essentials:

you can create operations to execute steps in your pipeline:

```
>>> from chariots.sklearn import SKUnsupervisedOp, SKSupervisedOp
>>> from chariots.versioning import VersionType, VersionedFieldDict, VersionedField
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
...
...
>>> class PCAOp(SKUnsupervisedOp):
...     training_update_version = VersionType.MAJOR
...     model_parameters = VersionedFieldDict(VersionType.MAJOR, {"n_components": 2})
...     model_class = VersionedField(PCA, VersionType.MAJOR)
...
>>> class LogisticOp(SKSupervisedOp):
...     training_update_version = VersionType.PATCH
...     model_class = LogisticRegression
```

Once your ops are created, you can create your various training and prediction pipelines:

```
>>> from chariots import Pipeline, MLMode
>>> from chariots.nodes import Node
...
...
>>> train = Pipeline([
...     Node(IrisFullDataSet(), output_nodes=["x", "y"]),
...     Node(PCAOp(MLMode.FIT_PREDICT), input_nodes=["x"], output_nodes="x_transformed
↪"),
...     Node(LogisticOp(MLMode.FIT), input_nodes=["x_transformed", "y"])
```

(continues on next page)

(continued from previous page)

```
... ], 'train')
...
>>> pred = Pipeline([
...     Node(PCAOp(MLMode.PREDICT), input_nodes=["__pipeline_input__"], output_nodes=
↪"x_transformed"),
...     Node(LogisticOp(MLMode.PREDICT), input_nodes=["x_transformed"], output_nodes=[
↪'__pipeline_output__'])
... ], 'pred')
```

Once all your pipelines have been created, deploying them is as easy as creating a creating a *Chariots* object:

```
>>> from chariots import Chariots
...
...
>>> app = Chariots([train, pred], app_path, import_name='iris_app')
```

The *Chariots* class inherits from the *Flask* class so you can deploy this the same way you would any flask application.

Once this the server is started, you can use the chariots client to query your machine learning micro-service from python:

```
>>> from chariots import Client
...
...
>>> client = Client()
```

with this client we will be

- training the models
- saving them and reloading the prediction pipeline (so that it uses the latest/trained version of our models)
- query some prediction

```
>>> client.call_pipeline(train)
>>> client.save_pipeline(train)
>>> client.load_pipeline(pred)
>>> client.call_pipeline(pred, [[1, 2, 3, 4]])
[1]
```

9.2 Features

- versionable individual op
- easy pipeline building
- easy pipelines deployment
- ML utils (implementation of ops for most popular ML libraries with adequate *Versionedfield*) for sklearn and keras at first
- A CookieCutter template to properly structure your Chariots project

9.3 Comming Soon

Some key features of Chariot are still in development and should be coming soon:

- Cloud integration (integration with cloud services to fetch and load models from)
- Graphql API to store and load information on different ops and pipelines (performance monitoring, . . .)
- ABTesting

9.4 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template. audreyr/cookiecutter-pypackage's project is also the basis of the Chariiots project template

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V