DLHub_SDK Documentation

Release 0.10.2

The Data and Learning Hub for Science

Jun 14, 2022

Contents:

DLHub SDK contains a Python interface to the Data and Learning Hub for Science (DLHub). These interfaces include functions for quickly describing a model in the correct schema for DLHub, and discovering or using models that other scientists have published.

Source Code: https://github.com/DLHub-Argonne/dlhub_sdk

CHAPTER 1

Quickstart

Welcome to the DLHub SDK. This SDK simplifies interacting with the DLHub service and the deployed servables.

1.1 Installation

The SDK is available on PyPI, but first make sure you have Python3.5+

```
>>> python3 --version
```

The CLI has been tested on Linux.

1.1.1 Installation using Pip

While `pip` and `pip3` can be used to install the CLI we suggest the following approach for reliable installation when many Python environments are available.:

```
    $ python3 -m pip install dlhub_sdk

To update a previously installed dlhub_sdk to a newer version, use: ``python3 -m pip␣
↪install -U dlhub_sdk)``
```

1.1.2 Installation using Conda

1. Install Conda and setup python3.6 following the instructions here:

   ```
   $ conda create --name dlhub_py36 python=3.6
   $ source activate dlhub_py36
   ```

2. Install the SDK:

```
$ python3 -m pip install dlhub_sdk

(to update a previously installed DLHub SDK to a newer version, use: python3 -m␣
↪pip install -U dlhub_sdk)
```

1.2 For Developers

1. Download the SDK:

```
$ git clone https://github.com/DLHub-Argonne/dlhub_sdk
```

2. Install:

```
$ cd dlhub_sdk
$ python3 setup.py install
```

3. Use the SDK!

1.3 Requirements

DLHub_SDK requires the following:

- Python 3.5+

CHAPTER 2

Servable Publication Guide

Publishing a new servable to DLHub requires two main steps: describing the servable and submitting it. In this guide, we provide a tutorial for describing different kinds of servables and an overview of the routes available for sending it to DLHub.

2.1 Step 1: Preparing the Submission Directory

You will first need to install the dlhub_cli and dlhub_sdk on your system. Both libraries are on PyPi, so you can install them with:

```
$ pip install dlhub_sdk dlhub_cli
```

The first step in the servable submission process is to authenticate with dlhub: `dlhub login`

Then, use the DLHub CLI to initialize the script for describing your servable:

```
$ dlhub init --author "Your, Name" "Your Affiliation" --name unique_name
```

The `init` command creates a Python script, `describe_servable.py`, which you edit to further describe the function of your servable. This command requires a name, which should be different than any other servable you have published to DLHub[1] and can optionally take other information to pre-populate your description script. Call `dlhub init --help` for full details.

2.2 Step 2: Describing a Servable

The `describe_servable.py` file created in Step 1 will generate a file, `dlhub.json`, that contains a short description of your servable. You will need to edit `describe_servable.py` file to add enough information for DLHub and other humans to use your servable.

[1] But can duplicate names from other users

2.2.1 Describing the Purpose and Provenance

There are a few places in the `describe_servable.py` file to capture a description of what the servable does. As noted by the "TODO" comments in the template, we recommend for your to write a descriptive title and short abstract about the servable as well as indicating other resources (e.g., papers) where users can find more information.

2.2.2 Describing the Computational Environment

There are two routes for specifying environment: defining Python libraries or using `repo2docker`.

Basic Route: Specifying Python Libraries

The computational environment for many servables can be defined by only the required Python packages. If this is true for your servable, edit the `describe_servable.py` file to include these requirements:

```
# Add a specific requirement without version information
model.add_requirement('library')

# Autodetect the version installed on your environment
model.add_requirement('library', 'detect')

# Get the latest version on PyPI
model.add_requirement('library', 'latest')

# Get a specific version
model.add_requirement('library', '1.0.0b')
```

Advanced: Using `repo2docker`

DLHub uses the configuration files specified by repo2docker to define a complicated computational environment. We refer you to the documentation for repo2docker to learn how to describe an environment. Once you have created the files, edit the `describe_servable.py` to point to the directory containing your configuration:

```
# Include repo2docker files in current directory
model.parse_repo2docker_configuration()

# Include repo2docker files in a different directory
model.parse_repo2docker_configuration('../another/path')
```

2.2.3 Describing the Model

DLHub supports numerous types of servables through different `BaseServableModel` tools. Each tool builds a description for a certain type of servable (e.g., Keras model) and has a corresponding class on the DLHub backend that uses the description to run your servable. Select the tool from the Servable Types documentation that matches your servable and follow the guide for describing it. Common examples include:

- Generic Python Functions
- Tensorflow
- Keras
- Scikit-Learn

Once finished, the `dlhub.json` file created by running `describe_servable.py` will have all the information needed by DLHub to run your model. You can re-run `describe_servable.py` to update the description if you ever make changes to the servable.

2.3 Step 3: Submitting the Model

There are several routes for submitting a servable to DLHub.

2.3.1 Recommended: Publishing Servable to Git Repository

Our recommended route for submitting servables to DLHub is to first publish the servable on a publicly-accessible git repository. You will need to add the `dlhub.json` file to your git repository along with all files mentioned in that servable description. For servables that require large data files (e.g., weights for deep learning models), we recommend using git-lfs to include them in the repository or *using Globus to submit the servable*.

After publishing all associated file to GitHub, use the DLHub CLI to publish the repository:

```
# Publish from the root folder of a git repository
$ dlhub publish --repository https://github.com/ryanchard/dlhub_publish_example

# Publish from another path within of a git repository
$ dlhub publish --repository https://github.com/ryanchard/dlhub_publish_example␣
↪another/path
```

Note: Publication from a non-root directory is still under development

2.3.2 Publication via Direct Upload

It is also possible to submit servables directly from your computer to DLHub via HTTP:

```
$ dlhub publish --local
```

This route is recommend for servables you do not want to share publicly and have small file sizes.

2.3.3 Publication via Globus

Note: This feature is under development

Globus Transfer is our preferred route for publishing servables with large numbers or sizes of files. To publish from your personal computer, you may need to first install a Globus endpoint. If you are publishing from a high-performance computing center, Globus may already be configured and available for use. In either case, you may need to determine the endpoint ID of the system holding your data (see Endpoint Management on Globus.org to find it). Then, submit your data via Globus using the CLI:

```
$ dlhub publish --globus --endpoint <your endpoint ID>
```

Alternatively, allow the CLI to attempt to determine the endpoint ID:

```
$ dlhub publish --globus
```

CHAPTER 3

Guide to DLHubClient

The DLHubClient provides a Python wrapper around the DLHub and funcX web services. In this part of the guide, we describe how to use the client to publish, discover, and use servables.

3.1 Authorization

DLHub uses GlobusAuth to control access to the web services and provide identities to each of our users. Before creating the DLHubClient, you must log in to Globus:

```
from dlhub_sdk.client import DLHubClient
client = DLHubClient()
```

This will initiate a Globus Auth flow for you to grant the client access to the DLHub service, Globus Search, OpenID, and the funcX service. The `DLHubClient` class stores your credentials in your home directory (`~/.dlhub/credentials/DLHub_Client_tokens.json`) so that you will only need to log in to Globus once when using the client or the DLHub CLI.

Call the `logout` function to remove access to DLHub from your system:

```
client.logout()
```

`logout` removes the credentials from your hard disk and revokes their authorization to prevent further use.

3.2 Publishing Servables

The DLHubClient provides several routes for publishing servables to DLHub. The first is to request DLHub to publish from a GitHub repository:

```
client.publish_repository('https://github.com/DLHub-Argonne/example-repo.git')
```

There are also functions for publishing a model stored on the local system. In either case, you must first create a `BaseServableModel` describing your servable or load in an existing one from disk:

```
from dlhub_sdk.utils import unserialize_object
with open('dlhub.json') as fp:
    model = unserialize_object(json.load(fp))
```

Then, submit the model via HTTP by calling:

```
client.publish_servable(model)
```

See the Publication Guide for details on how to describe a servable.

3.3 Discovering Servables

The client also provides tools for querying the library of servables available through DLHub.

The most general route for discovering models is to perform a free-text search of the model library:

```
client.search('"deep learning"')
```

You can also perform a query to match specific fields of the metadata records by setting the "advanced" flag on your query. For example, matching all Keras models related to materials science is accomplished by:

```
client.search('servable.type:"Keras Model" AND dlhub.domains:"chemistry"',␣
↪advanced=True)
```

See Globus Search documentation for complete information about the query string syntax and the DLHub schemas for the available query terms.

The client also provides functions for common queries, such as:

- search_by_authors to find servables from certain authors
- search_by_related_doi to find servables associated with a certain publication
- search_by_servable to find servables by their name or owner

Each of these tools returns metadata for only the most recent version of the servable by default, but can be configured to return all versions.

A way to perform advanced queries besides to craft your own query string is to use the "query helper" object that backs each of the pre-configured search functions. A new query helper is created by calling:

```
client.query
```

to return a blank query object, which you can then use to create a query with the advanced functions provided by the DLHubSearchHelper. For example, the advanced query shown above can be executed using:

```
client.query.match_term('servable.type', '"Keras Model"').match_domains('chemistry').
↪search()
```

3.4 Running Servables

The DLHubClient.run command runs servables published through DLHub. DLHub uses the funcX to perform remote inference tasks. funcX is a function-as-a-service platform designed to reliably and securely perform remote invocation.

Using the DLHub Client to perform an inference task invokes a funcX function on the target servable. To invoke the servable, you need to know the name of the servable:

```
client.run(model_name, x)
```

The data (`x`) will be serialized and passed to the servable to act on.

funcX performs client-side rate limiting to avoid accidentally overloading the service. By default, these limits restrict the number of invocations to 20 over 10 seconds and restrict input size to 5MB. These limits are described in detail in the funcX docs. The soft limits can be increased by modifying the DLHub client's funcX client::

```
client._fx_client.max_requests = 30
```

Note: The funcX web service will still restrict unnecessarily large inputs. Please contact us if this becomes an issue and we can help orchestrate big data invocations.

The DLHubClient.describe_servable and DLHubClient.describe_methods functions are especially useful when using an unfamiliar servable. The `describe_servable` method returns complete information about a servable, and the `describe_method` returns information about a certain method of the servable. Use these function to understand what the servable does and to learn how to use it.

CHAPTER 4

Servable Types

DLHub can serve many different kinds of functions and machine learning models. Each type of servable has a different tool (a "Model Class") that will aid you in collecting the data needed to run the servable. Here, we detail the types of servables available in DLHub and how to describe them.

4.1 Python Functions

It is possible to publish any Python function as a servable in DLHub. DLHub currently supports two types of Python functions: static functions and class methods. Static functions call members of Python modules and class methods involve calling a function of a specific Python object. Using `numpy` as an example, `numpy.sum(x)` involves calling a *static function* of the `numpy` module and `x.sum()` calls the `sum` *class method* of the `ndarray` `x`.

4.1.1 Python Static Functions

Model Class: PythonStaticMethodModel

Serving a Python function requires specifying the name of the module defining the function, the name of the function, and the inputs/outputs of the function. As an example, documenting the `max` function from `numpy` would start with:

```
model = PythonStaticMethodModel.create_model('numpy', 'max', autobatch=False,
                                             function_kwargs={'axis': 0})
```

The first arguments define the module and function name, and are followed by how the command is executed. `autobatch=True` would tell DLHub to run the function on each member of a list. `function_kwargs` defines the default keyword arguments for the function (in our case, `axis=0`)

The next step is to define the arguments to the function:

```
model.set_inputs('ndarray', 'Matrix', shape=[None, None])
model.set_outputs('ndarray', 'Max of a certain axis', shape=[None])
```

Each of these functions takes the type of input and a short description for that input. Certain types of inputs require further information (e.g., ndarrays require the shape of the array). See Argument Types for a complete listing of argument types.

Functions that take more than one argument (e.g., `f(x, y)`) require you to tell DLHub to unpack the inputs before running the function. As an example:

```
from dlhub_sdk.utils.types import compose_argument_block
model = PythonStaticMethodServable.from_function_pointer(f)
model.set_inputs('tuple', 'Two numbers', element_types=[
    compose_argument_block('float', 'A number'),
    compose_argument_block('float', 'A second number')
])
model.set_unpack_inputs(True)
```

Note that we used an input type of "tuple" to indicate that the function takes a fixed number of arguments. You can also use a type of "list" for functions that take a variable number of inputs.

Using Static Functions to Create Special Interfaces

Some servables required a specialized interface to make the software servable via DLHub. For example, some preprocessing of the input may need to occur before execution.

For these cases, we define a static function in file named `app.py` and create a Python servable for that interface function.

See our interface to `SchNet` as an example: link.

4.1.2 Python Class Method

Model Class: PythonClassMethodModel

Python class methods are functions associated with a specific Python object. DLHub needs both the file containing the object itself and documentation for the function. As an example, consider a Python object using the following code:

```
class ExampleClass:
    def __init__(self, a):
        self.a = a
    def f(self, x, b=1):
        return self.a * x + b
x = ExampleClass(1)
with open('pickle.pkl', 'wb') as fp:
    pkl.dump(x, fp)
```

The code to serve function `f` would be:

```
model = PythonClassMethodModel.create_model('pickle.pkl', 'f')
model.set_inputs('float', 'Input value')
model.set_outputs('float', 'Output value')
```

This code defines the file containing the serialized object (`pickle.pkl`), the name of the function to be run, and the types of the inputs and outputs. Note that the syntax for defining inputs and outputs is the same as the static functions.

For this example, it is necessary to include a module defining `ExampleClass` in the required libraries:

```
model.add_requirement('fake_module_with_exampleclass_on_pypi')
```

or adding the code that defines the class to a seperate file (e.g., `example.py`) and adding that to the list of files required by DLHub:

```
model.add_file('example.py')
```

As with the Python static methods, you can specify the functions with multiple arguments using `set_unpack_inputs`.

4.2 Keras Models

Model Class: KerasModel

DLHub serves Keras models using the HDF5 file saved using the `Model.save` function (see Keras FAQs). The methods described here also work with `tf.keras` though you should use the *Tensorflow* loader if you saved the model into Tensorflow's `SavedModel` format. As an example, the description for a Keras model created using:

```
model = Sequential()
model.add(Dense(16, input_shape=(1,), activation='relu', name='hidden'))
model.add(Dense(1, name='output'))
model.compile(optimizer='rmsprop', loss='mse')
model.fit(X, y)
model.save('model.h5')
```

can be generated from only the h5 model:

```
model_info = KerasModel.create_model('model.h5')
```

Models with weights and architecture as separate files can be described using:

```
model_info = KerasModel.create_model('model.h5', arch_path='arch.json')
```

Keras also allows users to add their own custom layers to their models for any custom operation that has trainable weights. Use this when the Keras Lambda layer does not apply. In Keras, these layers can be added when loading the model:

```
model = load_model('model.h5', custom_objects={'CustomLayer': CustomLayer})
```

Adding custom layers to a DLHub description can be achived with the `add_custom_object` method, which takes the name and class of the custom layer:

```
model_info.add_custom_object('CustomLayer', CustomLayer)
```

See more info on creating custom Keras layers here.

The DLHub SDK reads the architecture in the HDF5 file and determines the inputs and outputs automatically:

```
{
  "methods": {
    "run": {
      "input": {
        "type": "ndarray", "description": "Tensor", "shape": [null, 1]
      },
      "output": {
        "type": "ndarray", "description": "Tensor", "shape": [null, 1]
      },
      "parameters": {},
```

(continues on next page)

(continued from previous page)

```
      "method_details": {
        "method_name": "predict"
      }
    }
  }
}
```

We recommended changing the descriptions for the inputs and outputs from their default values:

```
model_info['servable']['methods']['run']['output']['description'] = 'Response'
```

but the model is ready to be served without any modifications.

The SDK also determines the version of Keras on your system, and saves that in the requirements.

4.3 PyTorch Models

Model Class: TorchModel

DLHub serves PyTorch models using the .pt file saved using the `torch.save` function (see PyTorch FAQs). As an example, the description for a PyTorch model created using:

```
class Net(nn.Module):
  def __init__(self):
    super(Net, self).__init__()
    self.conv1 = nn.Conv2d(1, 20, 5, 1)
    self.conv2 = nn.Conv2d(20, 50, 5, 1)
    self.fc1 = nn.Linear(4*4*50, 500)
    self.fc2 = nn.Linear(500, 10)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2, 2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2, 2)
    x = x.view(-1, 4*4*50)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return F.log_softmax(x, dim=1)

model = Net()
torch.save(model, 'model.pt')
```

can be generated from the .pt file and the shapes of the input and output arrays.

```
model_info = TorchModel.create_model('model.pt', (None, 1, 28, 28), (None, 10))
```

DLHub will need the definition for the `Net` module in order to load and run it. You must add the Python libraries containing the module definitions as requirements, or add the files defining the modules to the servable definition.

```
model_info.add_file('Net.py')
```

As with Keras, we recommended changing the descriptions for the inputs and outputs from their default values:

```
model_info['servable']['methods']['run']['output']['description'] = 'Response'
```

but the model is ready to be served without any modifications.

In some cases, you may need to specify the data types of your input array(s) using the keyword arguments of `create_model`. The type specifications are needed because PyTorch does not do type casting automatically. If in doubt, the data type is `float` and you can use the default settings.

The SDK also determines the version of Torch on your system, and saves that in the requirements.

4.4 TensorFlow Graphs

Model Class: TensorFlowModel

DLHub uses the same information as TensorFlow Serving for serving a TensorFlow model.

DLHub supports multiple functions to be defined for the same `SavedModel` servable, but requires one function is marked with `DEFAULT_SERVING_SIGNATURE_DEF_KEY`.

The SDK also determines the version of TensorFlow installed on your system, and lists it as a requirement.

How these models are created is very different between TF1 and TF2.

4.4.1 TF1

Save your model using the `SavedModelBuilder` as described in the TensorFlow v1.0. As an example, consider a graph expressing $y = x + 1$:

```
# Create the graph
with tf.Session() as sess:
    x = tf.placeholder('float', shape=(None, 3), name='Input')
    y = x + 1

    # Prepare to save the function
    builder = tf.saved_model.builder.SavedModelBuilder('./export')

    #  Make descriptions for the inputs and outputs
    x_desc = tf.saved_model.utils.build_tensor_info(x)
    y_desc = tf.saved_model.utils.build_tensor_info(y)

    # Create a function signature
    func_sig = tf.saved_model.signature_def_utils.build_signature_def(
        inputs={'x': x_desc},
        outputs={'y': y_desc},
        method_name='run'
    )

    # Add the session, graph, and function signature to the saved model
    builder.add_meta_graph_and_variables(
        sess, [tf.saved_model.tag_constants.SERVING],
        signature_def_map={
            tf.saved_model.signature_constants.DEFAULT_SERVING_SIGNATURE_DEF_KEY:␣
↪func_sig
        }
    )
```

(continues on next page)

(continued from previous page)

```
    # Write the files
    builder.save()
```

The DLHub SDK reads the `./export` directory written by this code:

```
metadata = TensorFlowModel.create_model("./export")
```

to generate metadata describing which functions were saved:

```
{
  "methods": {
    "run": {
      "input": {
        "type": "ndarray", "description": "x", "shape": [null, 3]
      },
      "output": {
        "type": "ndarray", "description": "y", "shape": [null, 3]
      },
      "parameters": {},
      "method_details": {
        "input_nodes": ["Input:0"],
        "output_nodes": ["add:0"]
      }
    }
  }
}
```

4.4.2 TF2

Follow the instructions in Tensorflow's documentation to save your model into the SavedModel format. DLHub requires you to specify the signatures for each of your function you wish to serve, which means you must either specify the input signature when defining the `tf.function` or create a concrete version of the function (see documentation).

The following example shows how to save a `tf.Module` with one function without a signature and a second with a signature.

```
class CustomModule(tf.Module):

    def __init__(self):
        super().__init__()
        self.m = tf.Variable([1.0, 1.0, 1.0], name='slope')

    @tf.function
    def __call__(self, x):
        y = self.m * x + 1
        return y

    @tf.function(input_signature=[tf.TensorSpec([], tf.float32),
                                  tf.TensorSpec((None, 3), tf.float32)])
    def scalar_multiply(self, z, x):
        return tf.multiply(z, x, name='scale_mult')

module = CustomModule()

# Make a concrete version of __call__
```

(continues on next page)

(continued from previous page)

```
call = module.__call__.get_concrete_function(tf.TensorSpec((None, 3)))

tf.saved_model.save(
    module, "./export", signatures={
        tf.saved_model.DEFAULT_SERVING_SIGNATURE_DEF_KEY: call,
        'scalar_multiply': module.scalar_multiply
    }
)
```

The DLHub SDK will automatically recognize the function signatures and use them to construct a servable accordingly:

```
metadata = TensorFlowModel.create_model("./export")
```

will generate metadata describing which functions were saved:

```
{
  "run": {
    "input": {
      "type": "ndarray", "description": "x:0", "shape": [null, 3],
      "item_type": {"type": "float"}
    },
    "output": {
      "type": "ndarray", "description": "Identity:0", "shape": [null, 3],
      "item_type": {"type": "float"}
    },
  }, "scalar_multiply": {
    "input": {
      "type": "tuple", "description": "Several tensors",
      "element_types": [{
          "type": "ndarray", "description": "x:0", "shape": [null, 3], "item_type": {
↪"type": "float"}
        }, {
          "type": "ndarray", "description": "z:0", "shape": [], "item_type": {"type":
↪"float"}
        }
      ]
    },
    "output": {
      "type": "ndarray", "description": "Identity:0", "shape": [null, 3],
      "item_type": {"type": "float"}
    },
  }
}
```

4.5 Scikit-Learn Models

Model Class: ScikitLearnModel

DLHub supports scikit-learn models saved using either pickle or joblib. The saved models files do not always contain the number of input features for the model, so they need to provided along with the serialization method and, for classifiers, the class names:

```
# Loading SVC trained on the iris dataset
model_info = ScikitLearnModel.create_model('model.pkl', n_input_columns=4, classes=3)
```

Given this information, the SDK generates documentation for how to invoke the model:

```
{
  "methods": {
    "run": {
      "input": {
        "type": "ndarray",
        "shape": [null, 4],
        "description": "List of records to evaluate with model. Each record is a list␣
↪of 4 variables.",
        "item_type": {"type": "float"}
      },
      "output": {
        "type": "ndarray",
        "shape": [null, 3],
        "description": "Probabilities for membership in each of 3 classes",
        "item_type": {"type": "float"}
      },
      "parameters": {},
      "method_details": {
        "method_name": "_predict_proba"
      }
    }
  }
}
```

The SDK will automatically document the type of model and extract the scikit-learn version used to save the model, which it includes in the requirements.

CHAPTER 5

Argument Types

DLHub supports many types of data as inputs and outputs to servables. In this part of the guide, we describe what these types are and how to define them when describing an interface. A full listing of the types is maintained in the DLHub schemas repository. A utility for creating type definitions can also be found in the DLHub SDK.

5.1 float, integer, number, complex

There are a variety of ways to express numerical values in DLHub interfaces: - `number` implies any real numerical value and implies there is limitation of the data being a float or integer. - `float` and `integer` are available if it is necessary to ensure the values are stored as floats or integers. - The `complex` argument type is used for complex numbers

5.2 string

Used for any string values.

5.3 boolean

Used for Boolean values.

5.4 timedelta, datetime

These data types define values associated with time. `timedelta` and `datetime` represent a length of time and a specific time, respectively.

5.5 python object

The `python object` type is used for data that cannot be expressed by other types. The only required argument for the `python object` is the Python type of the object by listed the Python class as `python_type` keyword. For example, a Pandas Dataframe would be expressed as:

```
{
    "type": "python object",
    "python_type": "pandas.DataFrame"
}
```

5.6 ndarray

`ndarray` values are matrices. It is required to specify the shape using the `shape` keyword, which takes a list of integers are `None` values. The `null` values in a shape definition represent that the dimension can take on any size. The type of each value in the array can be defined using the `item_type` keyword, which takes a type definition as its only argument. For example, an `Nx3` array of integers can be represented by:

```
{
   "type": "ndarray",
   "item_type": {"type": "integer"},
   "shape": [null, 3]
}
```

5.7 file

File inputs to a servable are supported by the `file` argument type. Users can specify what types are allowed using MIME types For example, an application that takes CSV files and Excel spreadsheets would represented as:

```
{
   "type": "file",
   "file_types": [
      "text/csv",
      "application/vnd.ms-excel",
      "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"
   ]
}
```

Files can only be mixed with other types of inputs in a reduced number of ways. The only acceptable types of function signatures with files are:

1. A single file is the sole input/output
2. A list of files is the sole input/output
3. Files or lists of files is one or more of the inputs or outputs to the function

In other words, files may not be part of a dictionary or a tuple data structure, or as a list of list of files.

5.8 list

List types define an ordered collection of indefinite length of all the same type of items. Only the item type need be defined using the `item_type` keyword, which requires an argument type as its value. For example, a list of 1D `ndarray` would be:

```
{
    "type": "list",
    "item_type": {
        "type": "ndarray",
        "shape": [null]
    }
}
```

5.9 tuple

Tuple types define an ordered collection of known length where each member can be a different type. The item type of each member and, thereby, the length must be defined using the `member_types` keyword. A tuple of a integer, float, and list of strings would be:

```
{
    "type": "tuple",
    "element_types": [
        {"type": "integer"},
        {"type": "float"},
        {"type": "list", "item_type": "string"}
    ]
}
```

5.10 dict

The `dict` argument type is used for dictionary objects. The data type requires the names and types of each key to be defined in the `properties` keyword. For example, a dictionary with key "x" mapped to an integer and "y" mapped to a float would be:

```
{
    "type": "dict",
    "properties": {
        "x": {"type": "integer"},
        "y": {"type": "float"}
    }
}
```

CHAPTER 6

dlhub_sdk

6.1 dlhub_sdk package

6.1.1 Subpackages

dlhub_sdk.models package

Subpackages

dlhub_sdk.models.servables package

Subpackages

Submodules

dlhub_sdk.models.servables.keras module

class dlhub_sdk.models.servables.keras.**KerasModel**
Bases: *dlhub_sdk.models.servables.python.BasePythonServableModel*

Servable based on a Keras Model object.

Assumes that the model has been saved to an hdf5 file

add_custom_object(*name*, *custom_object*)
Add a custom layer to the model specification

See *Keras FAQs <https://keras.io/getting-started/faq/#handling-custom-layers-or-other-custom-objects-in-saved-models>* for details.

Parameters

- **name** (*string*) – Name of the custom object

- **custom_object** (*class*) – Class of the custom object

Returns self

classmethod create_model(*model_path*, *output_names=None*, *arch_path=None*, *custom_objects=None*, *force_tf_keras: bool = False*) → dlhub_sdk.models.servables.keras.KerasModel

Initialize a Keras model.

Parameters

- **model_path** (*string*) – Path to the hd5 file that contains the weights and, optionally, the architecture
- **output_names** (*[string] or [[string]]*) – Names of output classes.
- **arch_path** (*string*) – Path to the hd5 model containing the architecture, if not available in the file at `model_path`.
- **custom_objects** (*dict*) – Map of layer names to custom layers. See Keras Documentation for more details.
- **force_tf_keras** (*bool*) – Force the use of TF.Keras even if keras is installed

format_layer_spec(*layers*) → dlhub_sdk.models.servables.ArgumentTypeMetadata

Make a description of a list of input or output layers

Parameters **layers** (*tuple or [tuple]*) – Shape of the layers

Returns (dict) Description of the inputs / outputs

dlhub_sdk.models.servables.pytorch module

class dlhub_sdk.models.servables.pytorch.**TorchModel**

Bases: *dlhub_sdk.models.servables.python.BasePythonServableModel*

Servable based on a Torch Model object.

Assumes that the model has been saved to a pt or a pth file

classmethod create_model(*model_path*, *input_shape*, *output_shape*, *input_type='float'*, *output_type='float'*)

Initialize a PyTorch model.

Parameters

- **model_path** (*string*) – Path to the pt or pth file that contains the weights and the architecture
- **input_shape** (*tuple or [tuple]*) – Shape of input matrix to model
- **output_shape** (*tuple or [tuple]*) – Shape of output matrix from model
- **input_type** (*str or [str]*) – Data type of inputs
- **output_type** (*str or [str]*) – Data type of outputs

format_layer_spec(*layers*, *datatypes*)

Make a description of a list of input or output layers

Parameters

- **layers** (*tuple or [tuple]*) – Shape of the layers
- **datatypes** (*str or [str]*) – Data type of each input layer

Returns (dict) Description of the inputs / outputs

dlhub_sdk.models.servables.python module

Tools to annotate generic operations (e.g., class method calls) in Python

class dlhub_sdk.models.servables.python.**BasePythonServableModel**
Bases: *dlhub_sdk.models.servables.BaseServableModel*

Describes a static python function to be run

classmethod create_model(*method*, *function_kwargs=None*) → dlhub_sdk.models.servables.python.BasePythonServableModel
Initialize a model for a python object

Parameters

- **method** (*string*) – Name of the method for this class
- **function_kwargs** (*dict*) – Names and values of any other argument of the function to set the values must be JSON serializable.

set_input_description(*description*, *method='run'*)
Set the human-readable description for this servable's inputs

This method can be called when implementing a Keras, PyTorch, etc. servable to fill in an empty input description.

Parameters

- **description** (*string*) – Human-readable description of the servable's inputs
- **method** (*string*) – Name of the servable method to apply description to (by default, 'run')

set_inputs(*data_type*, *description*, *shape=()*, *item_type=None*, ***kwargs*)
Define the inputs to the default ("run") function

Parameters

- **data_type** (*string*) – Type of the input data
- **description** (*string*) – Human-friendly description of the data
- **shape** (*list*) – Required for data_type of list or ndarray. Use *None* for dimensions that can have any numbers of values
- **item_type** (*string/dict*) – Description of the item type. Required for data_type = list

set_output_description(*description*, *method='run'*)
Set the human-readable description for this servable's inputs

This method can be called when implementing a Keras, PyTorch, etc. servable to fill in an empty input description.

Parameters

- **description** (*string*) – Human-readable description of the servable's inputs
- **method** (*string*) – Name of the servable method to apply description to (by default, 'run')

set_outputs(*data_type*, *description*, *shape=()*, *item_type=None*, ***kwargs*)
Define the outputs to the default ("run") function

Parameters

- **data_type** (*string*) – Type of the output data
- **description** (*string*) – Human-friendly description of the data
- **shape** (*list*) – Required for data_type of ndarray. Use *None* for dimensions that can have any numbers of values
- **item_type** (*string*) – Description of the type of item in a list

set_unpack_inputs(*x*, *method_name='run'*)

Define whether the inputs need to be unpacked before executing the function

Set to *true* if the function takes more than one input. Otherwise, the default is *False*

Parameters

- **x** (*bool*) – Desired setting
- **method_name** (*str*) – Name of the method to modify

Returns self

Return type (*BasePythonServableModel*)

class dlhub_sdk.models.servables.python.**PythonClassMethodModel**

Bases: *dlhub_sdk.models.servables.python.BasePythonServableModel*

Model for describing servables where the function to be performed is a method of a class.

To use this model, you must define the path to the pickled object and the function of that object to be called. Any additional libraries (beyond the standard libraries) required to run the library and their versions must also be specified. You may also specify any arguments of the class that should be set as defaults.

classmethod create_model(*path*, *method*, *function_kwargs=None*) → dlhub_sdk.models.servables.python.PythonClassMethodModel

Initialize a model for a python object

Parameters

- **path** (*string*) – Path to a pickled Python file
- **method** (*string*) – Name of the method for this class
- **function_kwargs** (*dict*) – Names and values of any other argument of the function to set the values must be JSON serializable.

class dlhub_sdk.models.servables.python.**PythonStaticMethodModel**

Bases: *dlhub_sdk.models.servables.python.BasePythonServableModel*

Model for a servable that calls a Python static method. Static methods can be called without any other context, unlike the class methods in PickledClassServableModelBase.

An example static method is the sqrt operation from numpy, *numpy.sqrt*. You can make a model of this function by calling `PythonStaticMethodModel.from_function_pointer(numpy.sqrt)`.

classmethod create_model(*module*, *method*, *autobatch=False*, *function_kwargs=None*)

Initialize the method

Parameters

- **module** (*string*) – Name of the module holding the function
- **method** (*string*) – Name of the method for this class
- **autobatch** (*bool*) – Whether to automatically run this function on a list of inputs. Calls `map(f, list)`

- **function_kwargs** (*dict*) – Names and values of any other argument of the function to set the values must be JSON serializable.

classmethod from_function_pointer(*f*, *autobatch=False*, *function_kwargs=None*)
Construct the module given a function pointer

Parameters

- **f** (*object*) – A function pointer
- **autobatch** (*bool*) – Whether to run function on an iterable of entries
- **function_kwargs** (*dict*) – Any default options for this function

dlhub_sdk.models.servables.sklearn module

class dlhub_sdk.models.servables.sklearn.**ScikitLearnModel**
Bases: *dlhub_sdk.models.servables.python.BasePythonServableModel*

Metadata for a scikit-learn machine learning model

This class is build assuming that the inputs to the model will be a list of lists of fixed lengths. Models that take different kinds of inputs (e.g., Pipelines that include text-processing steps, KernelRidge models with custom kernel functions) will are not yet supported.

classmethod create_model(*path*, *n_input_columns*, *classes=None*, *serialization_method='pickle'*)
Initialize a scikit-learn model

Parameters

- **path** (*string*) – Path to model file
- **n_input_columns** (*int*) – Number of input columns for the model
- **classes** (*Union[int, tuple]*) – For classification models, number of output classes or a list-like object with the names of the classes
- **serialization_method** (*string*) – Library used to serialize model

inspect_model(*model*)
Extract and store metadata that describes an ML model

Parameters **model** (*BaseEstimator*) – Model to be inspected

dlhub_sdk.models.servables.tensorflow module

Module contents

class dlhub_sdk.models.servables.**ArgumentTypeMetadata**
Bases: pydantic.main.BaseModel

Description of an input argument

class dlhub_sdk.models.servables.**BaseServableModel**
Bases: *dlhub_sdk.models.BaseMetadataModel*

Base class for servables. Holds the metadata for the object and how to create and run the servable object.

register_function(*name*, *inputs*, *outputs*, *parameters=None*, *method_details=None*)
Registers a new function to this servable

See `compose_argument_type` utility function for how to define the inputs and outputs to this function.

Parameters

- **name** (*string*) – Name of the function (e.g., "run")
- **inputs** (*dict*) – Description of inputs to the function
- **outputs** (*dict*) – Description of the outputs of the function
- **parameters** (*dict*) – Any additional parameters for the function and their default values
- **method_details** (*dict*) – Any options used when constructing a shim to run this function.

class dlhub_sdk.models.servables.**MethodMetadata**
Bases: pydantic.main.BaseModel

Metadata that describes each method

class dlhub_sdk.models.servables.**ServableMetadata**
Bases: pydantic.main.BaseModel

Metadata for servable objects.

Captures the information that describe how to run a servable.

class Config
Bases: object

extra = 'allow'

Submodules

dlhub_sdk.models.datasets module

dlhub_sdk.models.pipeline module

Module contents

This module contains tools for describing objects being published to DLHub.

class dlhub_sdk.models.**BaseMetadataModel**
Bases: pydantic.main.BaseModel

Base class for models describing objects published via DLHub

Covers information that goes in the `datacite` block of the metadata file and some of the DLHub block.

There are many kinds of MetadataModel classes that each describe a different kind of object. Each of these different types are created using the `create_model` operation (e.g., `KerasModel.create_model('model.h5')`), but have different arguments depending on the type of object. For example, TensorFlow models only require the directory created when saving the model for serving but scikit-learn models require the pickle file, how the pickle was created (e.g., with joblib), and how many input features it requires.

Once created, you will need to fill in additional details about the object to make it reusable. The MetadataModel classes attempt to learn as much about an object as possible automatically, but there is some information that must be provided by a human. To start, you must define a title and name for the object and are encouraged to provide an abstract describing the model and list any associated papers/websites that describe the model. You will fill plenty of examples for how to describe the models in the DLHub_containers repostiory. Some types of objects require data specific to their type (e.g., Python servables need a list of required packages). We encourage you to find examples for your specific type of object in the containers repository for inspiration and to see the Python documentation for each Metadata Model.

The MetadataModel object can be saved using the *to_dict* operation and read back into memory using the *from_dict* method. We recommend you save your dictionary to disk in the JSON or yaml format, which will allow for manual edits to be made before submitting or resubmitting a object description.

add_directory(*directory: str*, *include: Union[str, Iterable[str]] = ()*, *exclude: Union[str, Iterable[str]] = ()*, *recursive: bool = False*)

Add all the files in a directory

Parameters

- **include** (*string or [string]*) – Only add files that match any of these patterns
- **exclude** (*string or [string]*) – Exclude all files that match any of these patterns
- **directory** (*string*) – Path to a directory
- **recursive** (*bool*) – Whether to add all files in a directory

add_file(*file*, *name=None*)

Add a file to the list of files to be distributed with this object

Parameters

- **file** (*string*) – Path to the file
- **name** (*string*) – Optional. Name of the file, if it is a file that serves a specific purpose (e.g., "pickle" if this is a pickle file of a scikit-learn model)

add_files(*files: Iterable[str]*)

Add files that should be distributed with this artifact.

Parameters **files** – Paths of files that should be published

add_related_resource(*identifier: str, identifier_type: Union[str, dlhub_sdk.models.datacite.DataciteRelatedIdentifierType], relation_type: Union[str, dlhub_sdk.models.datacite.DataciteRelationType]*) → dlhub_sdk.models.BaseMetadataModel

Add a resource that is related to this object.

We use the DataCite to describe the relations. Common relation types for DLHub objects are:

- "IsDescribedBy": For a paper that describes a dataset or model
- "IsDocumentedBy": For the software documentation for a model
- "IsDerviedFrom": For the database a training set was pulled from
- "Requires": For any software libraries that are required for this module

Parameters

- **identifier** – Identifier
- **identifier_type** – Identifier type
- **relation_type** – Relation between this identifier and the object you are describing

add_requirement(*library*, *version=None*)
Add a required Python library.

The name of the library should be either the name on PyPI, or a URL for the git repository holding the code (e.g., `git+https://github.com/DLHub-Argonne/dlhub_sdk.git`)

Parameters

- **library** (*string*) – Name of library
- **version** (*string*) – Required version. 'latest' to use the most recent version on PyPi (if available). 'detect' will attempt to find the version of the library installed on the computer running this software. Default is `None`

add_requirements(*requirements*)
Add several Python library requirements

Parameters **requirements** (*dict*) – Keys are names of library (str), values are the version

classmethod create_model(***kwargs*) → dlhub_sdk.models.BaseMetadataModel
Instantiate the metadata model.

Takes in arguments that allow metadata describing a dataset to be autogenerated. For example, these could include options describing how to read a dataset from a CSV file or which class method to invoke on a Python pickle object.

get_zip_file(*path*)
Write all the listed files to a ZIP object

Takes all of the files returned by *list_files*. First determines the largest common path of all files, and preserves directory structure by using this common path as the root directory. For example, if the files are "/home/a.pkl" and "/home/a/b.dat", the common directory is "/home" and the files will be stored in the Zip as "a.pkl" and "a/b.dat"

Parameters **path** (*string*) – Path for the ZIP File

Returns

Base path for the ZIP file (useful for adjusting the paths of the files included in the metadata model)

Return type (string)

list_files()
Provide a list of files associated with this artifact.

Returns ([string]) list of file paths

name
Get the name of the servable

Returns (string) Name of the servable

parse_repo2docker_configuration(*directory=None*)
Gathers information about required environment from repo2docker configuration files.

See https://repo2docker.readthedocs.io/en/latest/config_files.html for more details

Parameters **directory** (*str*) – Path to directory containing configuration files (default: current working directory)

read_codemeta_file(*directory: Optional[str] = None*)
Read in metadata from a codemeta.json file

Parameters **directory** (*string*) – Path to directory contain the codemeta.json file (default: current working directory)

set_abstract (*abstract: str*) → dlhub_sdk.models.BaseMetadataModel
Define an abstract for this object. Use for a high-level summary

Parameters **abstract** – Description of this artifact

set_creators (*authors: List[str], affiliations: List[Sequence[str]] = ()*) → dlhub_sdk.models.BaseMetadataModel
Add authors to this object

Parameters

- **authors** – List of authors for the dataset. In format: "<Family Name>, <Given Name>"
- **affiliations** – List of affiliations for each author.

set_doi (*doi*)
Set the DOI of this object, if available

This function is only for advanced usage. Most users of the toolbox will not know the DOI before sending the object in to DLHub.

Parameters **doi** (*string*) – DOI of the object

set_domains (*domains: List[str]*)
Set the field of science that is associated with this object

Parameters **domains** – Names of fields of science (e.g., "materials science")

set_methods (*methods: str*) → dlhub_sdk.models.BaseMetadataModel
Define a methods section for this object. Use to describe any specific details about how the dataset, model, etc was generated.

Parameters **methods** (*str*) – Detailed method descriptions

set_name (*name*)
Set the name of the object.

Should be something short, descriptive, and memorable

Parameters **name** (*string*) – Name of artifact

set_publication_year (*year*)
Define the publication year

This function is only for advanced usage. Normally, this will be assigned automatically

Parameters **year** (*string*) – Publication year

set_title (*title: str*) → dlhub_sdk.models.BaseMetadataModel
Set the title for this object

Parameters **title** – Desired title

set_version (*version*)
Set the version of this resource

Parameters **version** (*string*) – Version number

set_visibility (*users: Optional[Iterable[str]] = None*, *groups: Optional[Iterable[str]] = None*)
Set the list of people this object should be visible to.

By default, it will be visible to anyone (["public"]).

Parameters

- **users** – GlobusAuth UUIDs of allowed users
- **groups** – GlobusAuth UUIDs of allowed Globus groups

to_dict(*simplify_paths: bool = False*)
Render the object to a JSON-ready dictionary

Parameters **simplify_paths** (*bool*) – Whether to simplify the paths of each file

Returns (dict) A description of the dataset in a form suitable for download

class dlhub_sdk.models.**DLHubMetadata**
Bases: pydantic.main.BaseModel

Basic metadata for a DLHub artefact

Includes information used by the DLHub web service to recognize an object, define how to build the computational environment, control access to it, et cetera

class dlhub_sdk.models.**DLHubType**
Bases: enum.Enum

Type supported by DLHub

dataset = 'dataset'

pipeline = 'piepline'

servable = 'servable'

dlhub_sdk.utils package

Submodules

dlhub_sdk.utils.schemas module

Utilities for validating against DLHub schemas

dlhub_sdk.utils.schemas.**validate_against_dlhub_schema**(*document: Union[dict, dlhub_sdk.models.BaseMetadataModel], schema_name: str*)
Validate a metadata document against one of the DLHub schemas

Note: Requires an internet connection

Parameters

- **document** – Document instance to be validated
- **schema_name** (*string*) – Name of schema (e.g., "dataset" for validating datasets). For full list, see: https://github.com/DLHub-Argonne/dlhub_schemas

Raises (jsonschema.SchemaError) If the schema fails to validate

dlhub_sdk.utils.types module

Utilities for generating descriptions of data types

dlhub_sdk.utils.types.**compose_argument_block**(*data_type*, *description*, *shape=None*, *item_type=None*, *python_type=None*, *properties=None*, *element_types=None*, ***kwargs*)

Compile a list of argument descriptions into an argument_type block

Parameters

- **data_type** (*string*) – Type of the input data
- **description** (*string*) – Human-friendly description of the data
- **shape** (*list*) – Required for data_type of list or ndarray. Use *None* for dimensions that can have any numbers of values
- **item_type** (*string/dict*) – Description of the item type. Required for data_type = list. Can either be a string type, or a dict that is a valid type for an argument type block
- **python_type** (*string*) – Full path of a Python object type (e.g., `pymatgen.core.Compostion`)
- **properties** (*dict*) – Descriptions of the types in a dictionary
- **element_types** (*[dict] or [str]*) – Types of elements in a tuple. List of type definition dictionaries or types as strings.

Keyword Arguments: Any other details particular to this kind of data :returns: (dict) Description of method in a form compatible with DLHub

dlhub_sdk.utils.types.**simplify_numpy_dtype**(*dtype*)

Given a numpy dtype, write out the type as string

Parameters **dtype** (*numpy.dtype*) – Type

Returns (string) name as a simple string

dlhub_sdk.utils.search module

Tools for interacting with the DLHub Search Index

class dlhub_sdk.utils.search.**DLHubSearchHelper**(*search_client: globus_sdk.services.search.client.SearchClient*, ***kwargs*)

Bases: `mdf_toolbox.globus_search.search_helper.SearchHelper`

Helper class for building queries with DLHub

match_authors(*authors*, *match_all=True*)

Add authors to the query.

Parameters

- **authors** (*str or list of str*) – The authors to match.
- **match_all** (*bool*) – If `True`, will require all authors be on any results. If `False`, will only require one author to be in results. **Default**: `True`.

Returns Self

Return type *DLHubSearchHelper*

match_doi(*doi*)

Add a DOI to the query.

Parameters **doi** (*str*) – The DOI to match.

Returns Self

Return type *DLHubSearchHelper*

match_domains(*domains*, *match_all=True*)
Add domains to the query.

Parameters

- **domains** (*str or list of str*) – The domains to match.
- **match_all** (*bool*) – If `True`, will require all domains be on any results. If `False`, will only require one domain to be in results. **Default**: `True`.

Returns Self

Return type *DLHubSearchHelper*

match_owner(*owner*)
Add a model owner to the query.

Parameters **owner** (*str*) – The name of the owner of the model.

Returns Self

Return type *DLHubSearchHelper*

match_servable(*servable_name=None*, *owner=None*, *publication_date=None*)
Add identifying model information to the query. If this method is called without any valid arguments, it will do nothing.

Parameters

- **servable_name** (*str*) – The name of the model. **Default**: None, to match all model names.
- **owner** (*str*) – The name of the owner of the model. **Default**: `None`, to match all owners.
- **publication_date** (*int*) – The UNIX timestamp for when the model was published. **Default**: `None`, to match all versions.

Returns Self

Return type *DLHubSearchHelper*

dlhub_sdk.utils.search.**filter_latest**(*results*)
Get only the models with the most recent publication date

Parameters **results** (*[dict]*) – List of results to filter

Returns Only the most recent results

Return type [dict]

dlhub_sdk.utils.search.**get_method_details**(*metadata*, *method_name=None*)
Get the method details for use by humans

Gets only the method fields out of the metadata record for an objecvt, removes the "method_details" field, which is used only during construction of the object.

Will either return the data for all methods or, if `method_name` is provided, only a single function

Parameters

- **metadata** (*dict*) – Metadata record for a servable

- **method_name** (*str*) – Optional: Name of the function to retrieve

Returns Metadata for the servable method, or a single method of `method_name` is used

Return type dict

Module contents

`dlhub_sdk.utils.`**unserialize_object**(*data*)

Given a metadata dictionary object, form a MetadataModel class

Parameters **data** (*dict*) – Metadata to unserialize

Returns (BaseMetadataModel) Unserialized object

6.1.2 Submodules

6.1.3 dlhub_sdk.client module

class `dlhub_sdk.client.`**DLHubClient**(*dlh_authorizer: Optional[globus_sdk.authorizers.base.GlobusAuthorizer] = None, search_authorizer: Optional[globus_sdk.authorizers.base.GlobusAuthorizer] = None, fx_authorizer: Optional[globus_sdk.authorizers.base.GlobusAuthorizer] = None, openid_authorizer: Optional[globus_sdk.authorizers.base.GlobusAuthorizer] = None, http_timeout: Optional[int] = None, force_login: bool = False, **kwargs*)

Bases: `globus_sdk.client.BaseClient`

Main class for interacting with the DLHub service

Holds helper operations for performing common tasks with the DLHub service. For example, *get_servables* produces a list of all servables registered with DLHub.

The initializer offers several routes for authenticating with the services:

1. **Token-based authentication. The default route for login is to log in with your own** Globus-associated account. Provided no arguments, the DLHub client will ask you to authenticate with Globus Auth then store your login tokens on your computer. The DLHub SDK will re-use these tokens unless you later call *logout()* or initialize the client with ‘‘force_login=True‘.
2. **Pre-defined authorizers**. The alternate route is to create authorizers using the Globus SDK directly

   and providing that authorizer to the initializer (e.g., `DLHubClient(dlhub_authorizer=auth)`). You must provide authorizers DLHub for all sub-services: Globus Search, FuncX, and OpenID.

clear_funcx_cache(*servable=None*)

Remove functions from the cache. Either remove a specific servable or wipe the whole cache.

Parameters **Servable** – str The name of the servable to remove. Default None

describe_methods(*name*, *method=None*)

Get the description for the method(s) of a certain servable

Parameters

- **name** (*string*) – DLHub name of the servable of the form <user>/<servable_name>
- **method** (*string*) – Optional: Name of the method

Returns

Description of a certain method if `method` provided, all methods if the method name was not provided.

Return type dict

`describe_servable`(*name*)
Get the description for a certain servable

Parameters **`name`** (*string*) – DLHub name of the servable of the form <user>/<servable_name>

Returns Summary of the servable

Return type dict

`get_result`(*task_id*, *verbose=False*)
Get the result of a task_id

Parameters

- **`str`** (*task_id*) – The task's uuid
- **`bool`** (*verbose*) – whether or not to return the full dlhub response

Returns Reult of running the servable

`get_servables`(*only_latest_version=True*)
Get all of the servables available in the service

Parameters **`only_latest_version`** (*bool*) – Whether to only return the latest version of each servable

Returns ([list]) Complete metadata for all servables found in DLHub

`get_task_status`(*task_id*)
Get the status of a DLHub task.

Parameters **`task_id`** (*string*) – UUID of the task

Returns status block containing "status" key.

Return type dict

`get_username`()
Get the username associated with the current credentials

`list_servables`()
Get a list of the servables available in the service

Returns List of all servable names in username/servable_name format

Return type [string]

`logout`()
Remove credentials from your local system

`publish_repository`(*repository*)
Submit a repository to DLHub for publication

Parameters **`repository`** (*string*) – Repository to publish

Returns Task ID of this submission, used for checking for success

Return type (string)

publish_servable(*model*)
Submit a servable to DLHub

If this servable has not been published before, it will be assigned a unique identifier.

If it has been published before (DLHub detects if it has an identifier), then DLHub will update the servable to the new version.

Parameters **model** (BaseMetadataModel) – Servable to be submitted

Returns Task ID of this submission, used for checking for success

Return type (string)

query
Access a query of the DLHub Search repository

run(*name: str*, *inputs: Any*, *parameters: Optional[Dict[str, Any]] = None*, *asynchronous: bool = False*, *debug: bool = False*, *async_wait: float = 5*, *timeout: Optional[float] = None*) → Union[dlhub_sdk.utils.futures.DLHubFuture, Tuple[Any, Dict[str, Any]], Any]
Invoke a DLHub servable

Parameters

- **name** – DLHub name of the servable of the form <user>/<servable_name>
- **inputs** – Data to be used as input to the function. Can be a string of file paths or URLs
- **parameters** – Any optional parameters to pass to the function.
- **asynchronous** – Whether to return from the function immediately or wait for the execution to finish.
- **debug** – Whether to capture the standard out and error printed during execution
- **async_wait** – How many seconds to wait between checking async status
- **timeout** – How long to wait for a result to return. Only used for synchronous calls

Returns

If asynchronous, a DLHubFuture for the execution If debug, the output of the function and dictionary holding the following information:

- success: Whether the code inside ran without raising an exception
- stdout/stderr: Captured standard output and error, if requested
- timing: Execution time for the segment in seconds
- exc: Captured exception object
- error_message: Exception traceback

If neither, the output of the function

run_serial(*servables*, *inputs*, *async_wait=5*)
Invoke each servable in a serial pipeline. This function accepts a list of servables and will run each one, passing the output of one as the input to the next.

Parameters

- **servables** (*list*) – A list of servable strings
- **inputs** – Data to pass to the first servable
- **asycn_wait** (*float*) – Seconds to wait between status checks

Returns Results of running the servable

search(*query*, *advanced=False*, *limit=None*, *only_latest=True*)
Query the DLHub servable library

By default, the query is used as a simple plaintext search of all model metadata. Optionally, you can provided an advanced query on any of the indexed fields in the DLHub model metadata by setting `advanced=True` and following the guide for constructing advanced queries found in the Globus Search documentation.

Parameters

- **query** (*string*) – Query to be performed
- **advanced** (*bool*) – Whether to perform an advanced query
- **limit** (*int*) – Maximum number of entries to return
- **only_latest** (*bool*) – Whether to return only the latest version of the model

Returns All records matching the search query

Return type ([dict])

search_by_authors(*authors*, *match_all=True*, *limit=None*, *only_latest=True*)
Execute a search for servables from certain authors.

Authors in DLHub may be different than the owners of the servable and generally are the people who developed functionality of a certain servable (e.g., the creators of the machine learning model used in a servable).

If you want to search by ownership, see *search_by_servable()*

Parameters

- **authors** (*str or list of str*) – The authors to match. Names must be in "Family Name, Given Name" format
- **match_all** (*bool*) – If `True`, will require all authors be on any results. If `False`, will only require one author to be in results. **Default**: `True`.
- **limit** (*int*) – The maximum number of results to return. **Default:** `None`, for no limit.
- **only_latest** (*bool*) – When `True`, will only return the latest version of each servable. When `False`, will return all matching versions. **Default**: `True`.

Returns List of servables from the desired authors

Return type [dict]

search_by_related_doi(*doi*, *limit=None*, *only_latest=True*)
Get all of the servables associated with a certain publication

Parameters

- **doi** (*string*) – DOI of related paper
- **limit** (*int*) – Maximum number of results to return
- **only_latest** (*bool*) – Whether to return only the most recent version of the model

Returns List of servables from the requested paper

Return type [dict]

search_by_servable(*servable_name=None*, *owner=None*, *version=None*, *only_latest=True*, *limit=None*, *get_info=False*)
Search by the ownership, name, or version of a servable

Parameters

- **servable_name** (*str*) – The name of the servable. **Default**: None, to match all servable names.
- **owner** (*str*) – The name of the owner of the servable. **Default**: `None`, to match all owners.
- **version** (*int*) – Model version, which corresponds to the date when the servable was published. **Default**: `None`, to match all versions.
- **only_latest** (*bool*) – When `True`, will only return the latest version of each servable. When `False`, will return all matching versions. **Default**: `True`.
- **limit** (*int*) – The maximum number of results to return. **Default:** `None`, for no limit.
- **get_info** (*bool*) – If `False`, search will return a list of the results. If `True`, search will return a tuple containing the results list and other information about the query. **Default:** `False`.

Returns The search results. If `info` is `True`, *tuple*: The search results, and a dictionary of query information.

Return type If `info` is `False`, *list*

service_name = 'DLHub'

6.1.4 Module contents

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

S

T

U

V