MLServer

Release 1.3.5

Seldon Technologies

Jul 10, 2023

CONTENTS

On this section you can learn more about the different features of MLServer and how to use them.

CHAPTER

ONE

CONTENT TYPES (AND CODECS)

Machine learning models generally expect their inputs to be passed down as a particular Python type. Most commonly, this type ranges from *"general purpose"* NumPy arrays or Pandas DataFrames to more granular definitions, like `datetime` objects, `Pillow` images, etc. Unfortunately, the definition of the V2 Inference Protocol doesn't cover any of the specific use cases. This protocol can be thought of a wider *"lower level"* spec, which only defines what fields a payload should have.

To account for this gap, MLServer introduces support for **content types**, which offer a way to let MLServer know how it should *"decode"* V2-compatible payloads. When shaped in the right way, these payloads should *"encode"* all the information required to extract the higher level Python type that will be required for a model.

To illustrate the above, we can think of a Scikit-Learn pipeline, which takes in a Pandas DataFrame and returns a NumPy Array. Without the use of **content types**, the V2 payload itself would probably lack information about how this payload should be treated by MLServer Likewise, the Scikit-Learn pipeline wouldn't know how to treat a raw V2 payload. In this scenario, the use of content types allows us to specify information on what's the actual *"higher level"* information encoded within the V2 protocol payloads.

1.1 Usage

Note: Some inference runtimes may apply a content type by default if none is present. To learn more about each runtime's defaults, please check the *relevant inference runtime's docs*.

To let MLServer know that a particular payload must be decoded / encoded as a different Python data type (e.g. NumPy Array, Pandas DataFrame, etc.), you can specifity it through the `content_type` field of the `parameters` section of your request.

As an example, we can consider the following dataframe, containing two columns: Age and First Name.

First Name	Age
Joanne	34
Michael	22

This table, could be specified in the V2 protocol as the following payload, where we declare that:

- The whole set of inputs should be decoded as a Pandas Dataframe (i.e. setting the content type as `pd`).
- The First Name column should be decoded as a UTF-8 string (i.e. setting the content type as `str`).

```
{
  "parameters": {
    "content_type": "pd"
  },
  "inputs": [
    {
      "name": "First Name",
      "datatype": "BYTES",
      "parameters": {
        "content_type": "str"
      },
      "shape": [2],
      "data": ["Joanne", "Michael"]
    },
    {
      "name": "Age",
      "datatype": "INT32",
      "shape": [2],
      "data": [34, 22]
    },
  ]
}
```

To learn more about the available content types and how to use them, you can see all the available ones in the *Available Content Types* section below.

Note: It's important to keep in mind that content types can be specified at both the **request level** and the **input level**. The former will apply to the **entire set of inputs**, whereas the latter will only apply to a **particular input** of the payload.

1.1.1 Codecs

Under the hood, the conversion between content types is implemented using *codecs*. In the MLServer architecture, codecs are an abstraction which know how to *encode* and *decode* high-level Python types to and from the V2 Inference Protocol.

Depending on the high-level Python type, encoding / decoding operations may require access to multiple input or output heads. For example, a Pandas Dataframe would need to aggregate all of the input-/output-heads present in a V2 Inference Protocol response.

However, a Numpy array or a list of strings, could be encoded directly as an input head within a larger request.

To account for this, codecs can work at either the request- / response-level (known as **request codecs**), or the input- / output-level (known as **input codecs**). Each of these codecs, expose the following **public interface**, where `Any` represents a high-level Python datatype (e.g. a Pandas Dataframe, a Numpy Array, etc.):

- **Request Codecs**
  - *encode_request()*
  - *decode_request()*

  - *encode_response()*
  - *decode_response()*
- **Input Codecs**
  - *encode_input()*
  - *decode_input()*
  - *encode_output()*
  - *decode_output()*

Note that, these methods can also be used as helpers to **encode requests and decode responses on the client side**. This can help to abstract away from the user most of the details about the underlying structure of V2-compatible payloads.

For example, in the example above, we could use codecs to encode the DataFrame into a V2-compatible request simply as:

```
import pandas as pd

from mlserver.codecs import PandasCodec

dataframe = pd.DataFrame({'First Name': ["Joanne", "Michael"], 'Age': [34, 22]})

inference_request = PandasCodec.encode_request(dataframe)
print(inference_request)
```

For a full end-to-end example on how content types and codecs work under the hood, feel free to check out this *Content Type Decoding example*.

Converting to / from JSON

When using MLServer's request codecs, the output of encoding payloads will always be one of the classes within the `mlserver.types` package (i.e. *InferenceRequest* or *InferenceResponse*). Therefore, if you want to use them with `requests` (or other package outside of MLServer) you will need to **convert them to a Python dict or a JSON string**.

Luckily, these classes leverage Pydantic under the hood. Therefore you can just call the `.dict()` or `.json()` method to convert them. Likewise, to read them back from JSON, we can always pass the JSON fields as kwargs to the class' constructor (or use any of the other methods available within Pydantic).

For example, if we want to send an inference request to model `foo`, we could do something along the following lines:

```
import pandas as pd
import requests

from mlserver.codecs import PandasCodec

dataframe = pd.DataFrame({'First Name': ["Joanne", "Michael"], 'Age': [34, 22]})

inference_request = PandasCodec.encode_request(dataframe)

# raw_request will be a Python dictionary compatible with `requests`'s `json` kwarg
raw_request = inference_request.dict()

response = requests.post("localhost:8080/v2/models/foo/infer", json=raw_request)
```

(continues on next page)

(continued from previous page)

```
# raw_response will be a dictionary (loaded from the response's JSON),
# therefore we can pass it as the InferenceResponse constructors' kwargs
raw_response = response.json()
inference_response = InferenceResponse(**raw_response)
```

1.1.2 Model Metadata

Content types can also be defined as part of the *model's metadata*. This lets the user pre-configure what content types should a model use by default to decode / encode its requests / responses, without the need to specify it on each request.

For example, to configure the content type values of the *example above*, one could create a `model-settings.json` file like the one below:

Listing 1: model-settings.json

```
{
  "parameters": {
    "content_type": "pd"
  },
  "inputs": [
    {
      "name": "First Name",
      "datatype": "BYTES",
      "parameters": {
        "content_type": "str"
      },
      "shape": [-1],
    },
    {
      "name": "Age",
      "datatype": "INT32",
      "shape": [-1],
    },
  ]
}
```

It's important to keep in mind that content types passed explicitly as part of the request will always **take precedence over the model's metadata**. Therefore, we can leverage this to override the model's metadata when needed.

1.2 Available Content Types

Out of the box, MLServer supports the following list of content types. However, this can be extended through the use of 3rd-party or custom runtimes.

Python Type	Content Type	Request Level	Request Codec	Input Level	Input Codec
NumPy Array	`np`		`mlserver.codecs.NumpyRequestCodec`		`mlserver.codecs.NumpyCodec`
Pandas DataFrame	`pd`		`mlserver.codecs.PandasCodec`		
UTF-8 String	`str`		`mlserver.codecs.string.StringRequestCodec`		`mlserver.codecs.StringCodec`
Base64	`base64`				`mlserver.codecs.Base64Codec`
Datetime	`datetime`				`mlserver.codecs.DatetimeCodec`

Note: MLServer allows you extend the supported content types by **adding custom ones**. To learn more about how to write your own custom content types, you can check this *full end-to-end example*. You can also learn more about building custom extensions for MLServer on the *Custom Inference Runtime section* of the docs.

1.2.1 NumPy Array

Note: The V2 Inference Protocol expects that the `data` of each input is sent as a **flat array**. Therefore, the `np` content type will expect that tensors are sent flattened. The information in the `shape` field will then be used to reshape the vector into the right dimensions.

The `np` content type will decode / encode V2 payloads to a NumPy Array, taking into account the following:

- The `datatype` field will be matched to the closest NumPy `dtype`.
- The `shape` field will be used to reshape the flattened array expected by the V2 protocol into the expected tensor shape.

Note: By default, MLServer will always assume that an array with a single-dimensional shape, e.g. `[N]`, is equivalent to `[N, 1]`. That is, each entry will be treated like a single one-dimensional data point (i.e. instead of a `[1, D]` array, where the full array is a single `D`-dimensional data point). To avoid any ambiguity, where possible, the **Numpy codec will always explicitly encode `[N]` arrays as `[N, 1]`**.

For example, if we think of the following NumPy Array:

```
import numpy as np

foo = np.array([[1, 2], [3, 4]])
```

We could encode it as the input `foo` in a V2 protocol request as:

JSON payload

```
{
  "inputs": [
    {
      "name": "foo",
      "parameters": {
        "content_type": "np"
      },
      "data": [1, 2, 3, 4]
      "datatype": "INT32",
      "shape": [2, 2],
    }
  ]
}
```

NumPy Request Codec

```
from mlserver.codecs import NumpyRequestCodec

# Encode an entire V2 request
inference_request = NumpyRequestCodec.encode_request(foo)
```

NumPy Input Codec

```
from mlserver.types import InferenceRequest
from mlserver.codecs import NumpyCodec

# We can use the `NumpyCodec` to encode a single input head with name `foo`
# within a larger request
inference_request = InferenceRequest(
  inputs=[
    NumpyCodec.encode_input("foo", foo)
  ]
)
```

When using the NumPy Array content type at the **request-level**, it will decode the entire request by considering only the first `input` element. This can be used as a helper for models which only expect a single tensor.

1.2.2 Pandas DataFrame

Note: The `pd` content type can be *stacked* with other content types. This allows the user to use a different set of content types to decode each of the columns.

The `pd` content type will decode / encode a V2 request into a Pandas DataFrame. For this, it will expect that the DataFrame is shaped in a **columnar way**. That is,

- Each entry of the `inputs` list (or `outputs`, in the case of responses), will represent a column of the DataFrame.

- Each of these entires, will contain all the row elements for that particular column.
- The `shape` field of each `input` (or `output`) entry will contain (at least) the amount of rows included in the dataframe.

For example, if we consider the following dataframe:

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4

We could encode it to the V2 Inference Protocol as:

JSON Payload

```
{
  "parameters": {
    "content_type": "pd"
  },
  "inputs": [
    {
      "name": "A",
      "data": ["a1", "a2", "a3", "a4"]
      "datatype": "BYTES",
      "shape": [3],
    },
    {
      "name": "B",
      "data": ["b1", "b2", "b3", "b4"]
      "datatype": "BYTES",
      "shape": [3],
    },
    {
      "name": "C",
      "data": ["c1", "c2", "c3", "c4"]
      "datatype": "BYTES",
      "shape": [3],
    },
  ]
}
```

Pandas Request Codec

```
import pandas as pd

from mlserver.codecs import PandasCodec

foo = pd.DataFrame({
  "A": ["a1", "a2", "a3", "a4"],
  "B": ["b1", "b2", "b3", "b4"],
  "C": ["c1", "c2", "c3", "c4"]
})

inference_request = PandasCodec.encode_request(foo)
```

1.2.3 UTF-8 String

The `str` content type lets you encode / decode a V2 input into a UTF-8 Python string, taking into account the following:

- The expected `datatype` is `BYTES`.
- The `shape` field represents the number of "strings" that are encoded in the payload (e.g. the `["hello world", "one more time"]` payload will have a shape of 2 elements).

For example, when if we consider the following list of strings:

```
foo = ["bar", "bar2"]
```

We could encode it to the V2 Inference Protocol as:

JSON Payload

```
{
  "parameters": {
    "content_type": "str"
  },
  "inputs": [
    {
      "name": "foo",
      "data": ["bar", "bar2"]
      "datatype": "BYTES",
      "shape": [2],
    }
  ]
}
```

String Request Codec

```
from mlserver.codecs.string import StringRequestCodec

# Encode an entire V2 request
inference_request = StringRequestCodec.encode_request(foo, use_bytes=False)
```

String Input Codec

```
from mlserver.types import InferenceRequest
from mlserver.codecs import StringCodec

# We can use the `StringCodec` to encode a single input head with name `foo`
# within a larger request
inference_request = InferenceRequest(
  inputs=[
    StringCodec.encode_input("foo", foo, use_bytes=False)
  ]
)
```

When using the `str` content type at the request-level, it will decode the entire request by considering only the first `input` element. This can be used as a helper for models which only expect a single string or a set of strings.

1.2.4 Base64

The `base64` content type will decode a binary V2 payload into a Base64-encoded string (and viceversa), taking into account the following:

- The expected `datatype` is `BYTES`.
- The `data` field should contain the base64-encoded binary strings.
- The `shape` field represents the number of binary strings that are encoded in the payload.

For example, if we think of the following *"bytes array"*:

```
foo = b"Python is fun"
```

We could encode it as the input `foo` of a V2 request as:

JSON Payload

```
{
  "inputs": [
    {
      "name": "foo",
      "parameters": {
        "content_type": "base64"
      },
      "data": ["UHl0aG9uIGlzIGZ1bg=="]
      "datatype": "BYTES",
      "shape": [1],
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

Base64 Input Codec

```
from mlserver.types import InferenceRequest
from mlserver.codecs import Base64Codec

# We can use the `Base64Codec` to encode a single input head with name `foo`
# within a larger request
inference_request = InferenceRequest(
  inputs=[
    Base64Codec.encode_input("foo", foo, use_bytes=False)
  ]
)
```

1.2.5 Datetime

The `datetime` content type will decode a V2 input into a Python `datetime.datetime` object, taking into account the following:

- The expected `datatype` is `BYTES`.
- The `data` field should contain the dates serialised following the ISO 8601 standard.
- The `shape` field represents the number of datetimes that are encoded in the payload.

For example, if we think of the following `datetime` object:

```
import datetime

foo = datetime.datetime(2022, 1, 11, 11, 0, 0)
```

We could encode it as the input `foo` of a V2 request as:

JSON Payload

```
{
  "inputs": [
    {
      "name": "foo",
      "parameters": {
        "content_type": "datetime"
      },
      "data": ["2022-01-11T11:00:00"]
      "datatype": "BYTES",
      "shape": [1],
    }
  ]
}
```

Datetime Input Codec

```
from mlserver.types import InferenceRequest
from mlserver.codecs import DatetimeCodec

# We can use the `DatetimeCodec` to encode a single input head with name `foo`
# within a larger request
inference_request = InferenceRequest(
  inputs=[
    DatetimeCodec.encode_input("foo", foo, use_bytes=False)
  ]
)
```

CHAPTER

TWO

OPENAPI SUPPORT

MLServer follows the Open Inference Protocol (previously known as the "V2 Protocol"). You can find the full OpenAPI spec for the Open Inference Protocol in the links below:

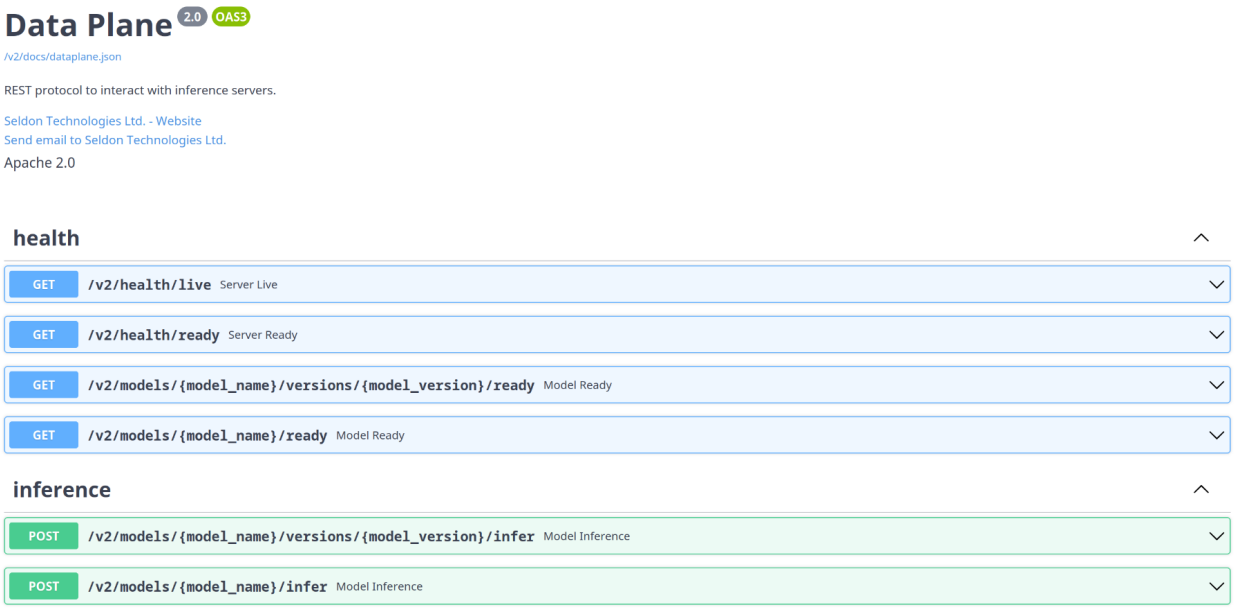
Name	Description	OpenAPI Spec
Open Inference Protocol	Main dataplane for inference, health and metadata	`dataplane.json`
Model Repository Extension	Extension to the protocol to provide a control plane which lets you load / unload models dynamically	`model_repository.json`

2.1 Swagger UI

On top of the OpenAPI spec above, MLServer also autogenerates a Swagger UI which can be used to interact dynamycally with the Open Inference Protocol.

The autogenerated Swagger UI can be accessed under the `/v2/docs` endpoint.

Note: Besides the Swagger UI, you can also access the *raw* OpenAPI spec through the `/v2/docs/dataplane.json` endpoint.



2.2 Model Swagger UI

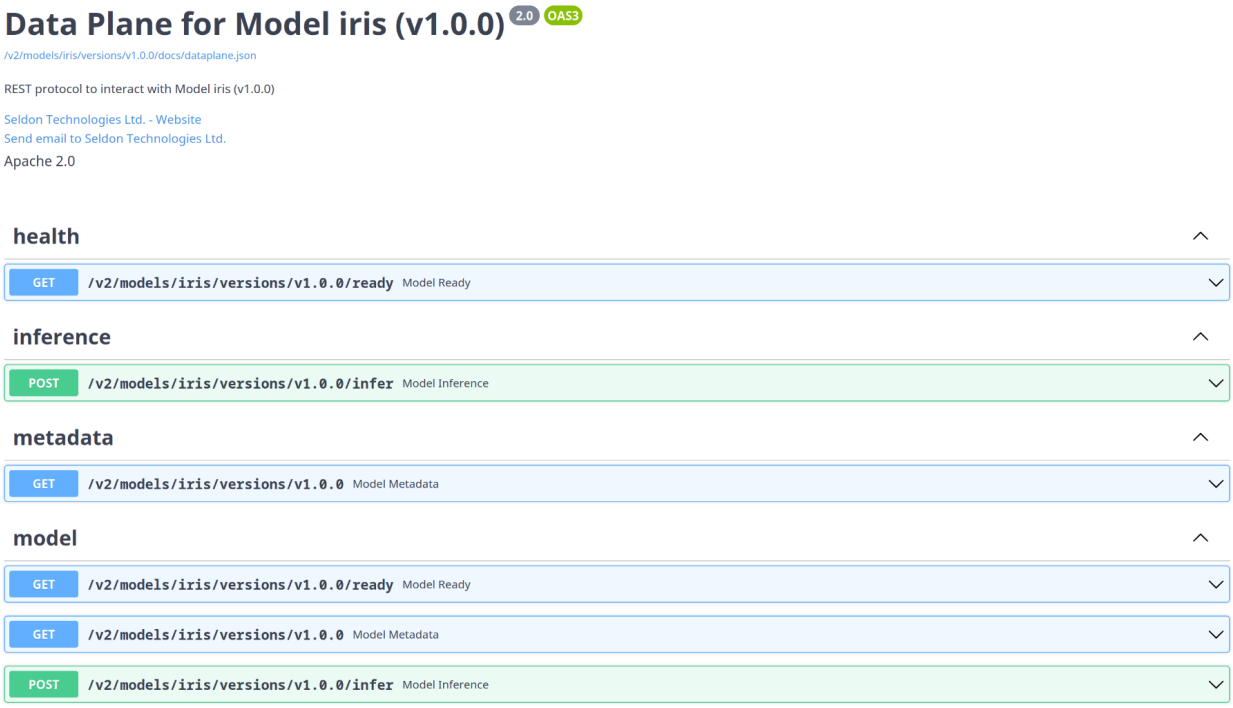
Alongside the *general API documentation*, MLServer will also autogenerate a Swagger UI tailored to individual models, showing the endpoints available for each one.

The model-specific autogenerated Swagger UI can be accessed under the following endpoints:

- `/v2/models/{model_name}/docs`
- `/v2/models/{model_name}/versions/{model_version}/docs`

Note: Besides the Swagger UI, you can also access the model-specific *raw* OpenAPI spec through the following endpoints:

- `/v2/models/{model_name}/docs/dataplane.json`
- `/v2/models/{model_name}/versions/{model_version}/docs/dataplane.json`


Data Plane for Model iris (v1.0.0)
2.0
OAS3
/v2/models/iris/versions/v1.0.0/docs/dataplane.json
REST protocol to interact with Model iris (v1.0.0)
Seldon Technologies Ltd. - Website
Send email to Seldon Technologies Ltd.
Apache 2.0
health
GET /v2/models/iris/versions/v1.0.0/ready Model Ready
inference
POST /v2/models/iris/versions/v1.0.0/infer Model Inference
metadata
GET /v2/models/iris/versions/v1.0.0 Model Metadata
model
GET /v2/models/iris/versions/v1.0.0/ready Model Ready
GET /v2/models/iris/versions/v1.0.0 Model Metadata
POST /v2/models/iris/versions/v1.0.0/infer Model Inference

CHAPTER

THREE

PARALLEL INFERENCE

Out of the box, MLServer includes support to offload inference workloads to a pool of workers running in separate processes. This allows MLServer to scale out beyond the limitations of the Python interpreter. To learn more about why this can be beneficial, you can check the *concurrency section* below.

By default, MLServer will spin up a pool with only one worker process to run inference. All models will be loaded uniformly across the inference pool workers. To read more about advanced settings, please see the *usage section below*.

3.1 Concurrency in Python

The Global Interpreter Lock (GIL) is a mutex lock that exists in most Python interpreters (e.g. CPython). Its main purpose is to lock Python's execution so that it only runs on a single processor at the same time. This simplifies certain things to the interpreter. However, it also adds the limitation that a **single Python process will never be able to leverage multiple cores**.

When we think about MLServer's support for *Multi-Model Serving (MMS)*, this could lead to scenarios where a **heavily-used model starves the other models** running within the same MLServer instance. Similarly, even if we don't take MMS into account, the **GIL also makes it harder to scale inference for a single model**.

To work around this limitation, MLServer offloads the model inference to a pool of workers, where each worker is a separate Python process (and thus has its own separate GIL). This means that we can get full access to the underlying hardware.

3.1.1 Overhead

Managing the Inter-Process Communication (IPC) between the main MLServer process and the inference pool workers brings in some overhead. Under the hood, MLServer uses the `multiprocessing` library to implement the distributed processing management, which has been shown to offer the smallest possible overhead when implementating these type of distributed strategies [Zhi *et al.*, 2020].

The extra overhead introduced by other libraries is usually brought in as a trade off in exchange of other advanced features for complex distributed processing scenarios. However, MLServer's use case is simple enough to not require any of these.

Despite the above, even though this overhead is minimised, this **it can still be particularly noticeable for lightweight inference methods**, where the extra IPC overhead can take a large percentage of the overall time. In these cases (which can only be assessed on a model-by-model basis), the user has the option to *disable the parallel inference feature*.

For regular models where inference can take a bit more time, this overhead is usually offset by the benefit of having multiple cores to compute inference on.

3.2 Usage

By default, MLServer will always create an inference pool with one single worker. The number of workers (i.e. the size of the inference pool) can be adjusted globally through the server-level `parallel_workers` setting.

3.2.1 `parallel_workers`

The `parallel_workers` field of the `settings.json` file (or alternatively, the `MLSERVER_PARALLEL_WORKERS` global environment variable) controls the size of MLServer's inference pool. The expected values are:

- `N`, where `N > 0`, will create a pool of `N` workers.
- `0`, will disable the parallel inference feature. In other words, inference will happen within the main MLServer process.

3.3 References

CHAPTER

FOUR

ADAPTIVE BATCHING

MLServer includes support to batch requests together transparently on-the-fly. We refer to this as "adaptive batching", although it can also be known as "predictive batching".

4.1 Benefits

There are usually two main reasons to adopt adaptive batching:

- **Maximise resource usage**. Usually, inference operations are "vectorised" (i.e. are designed to operate across batches). For example, a GPU is designed to operate on multiple data points at the same time. Therefore, to make sure that it's used at maximum capacity, we need to run inference across batches.
- **Minimise any inference overhead**. Usually, all models will have to *"pay"* a constant overhead when running any type of inference. This can be something like IO to communicate with the GPU or some kind of processing in the incoming data. Up to a certain size, this overhead tends to not scale linearly with the number of data points. Therefore, it's in our interest to send as large batches as we can without deteriorating performance.

However, these benefits will usually scale only up to a certain point, which is usually determined by either the infrastructure, the machine learning framework used to train your model, or a combination of both. Therefore, to maximise the performance improvements brought in by adaptive batching it will be important to *configure it with the appropriate values for your model*. Since these values are usually found through experimentation, **MLServer won't enable by default adaptive batching on newly loaded models**.

4.2 Usage

MLServer lets you configure adaptive batching independently for each model through two main parameters:

- **Maximum batch size**, that is how many requests you want to group together.
- **Maximum batch time**, that is how much time we should wait for new requests until we reach our maximum batch size.

4.2.1 max_batch_size

The `max_batch_size` field of the `model-settings.json` file (or alternatively, the `MLSERVER_MODEL_MAX_BATCH_SIZE` global environment variable) controls the maximum number of requests that should be grouped together on each batch. The expected values are:

- `N`, where `N > 1`, will create batches of up to `N` elements.
- `0` or `1`, will disable adaptive batching.

4.2.2 max_batch_time

The `max_batch_time` field of the `model-settings.json` file (or alternatively, the `MLSERVER_MODEL_MAX_BATCH_TIME` global environment variable) controls the time that MLServer should wait for new requests to come in until we reach our maximum batch size.

The expected format is in seconds, but it will take fractional values. That is, 500ms could be expressed as `0.5`.

The expected values are:

- `T`, where `T > 0`, will wait `T` seconds at most.
- `0`, will disable adaptive batching.

4.2.3 Merge and split of custom paramters

MLserver allows adding custom parameters to the `parameters` field of the requests. These parameters are recived as a merged list of parameters inside the server, e.g.

```
# request 1
types.RequestInput(
    name="parameters-np",
    shape=[1],
    datatype="BYTES",
    data=[],
    parameters=types.Parameters(
        custom-param='value-1',
    )
)

# request 2
types.RequestInput(
    name="parameters-np",
    shape=[1],
    datatype="BYTES",
    data=[],
    parameters=types.Parameters(
        custom-param='value-2',
    )
)
```

is recived as follows in the batched request in the server:

```
types.RequestInput(
    name="parameters-np",
    shape=[2],
    datatype="BYTES",
    data=[],
    parameters=types.Parameters(
        custom-param=['value-1', 'value-2'],
    )
)
```

The same way if the request is sent back from the server as a batched request

```
types.ResponseOutput(
    name="foo",
    datatype="INT32",
    shape=[3, 3],
    data=[1, 2, 3, 4, 5, 6, 7, 8, 9],
    parameters=types.Parameters(
        content_type="np",
        foo=["foo_1", "foo_2"],
        bar=["bar_1", "bar_2", "bar_3"],
    ),
)
```

it will be returned unbatched from the server as follows:

```
# Request 1
types.ResponseOutput(
    name="foo",
    datatype="INT32",
    shape=[1, 3],
    data=[1, 2, 3],
    parameters=types.Parameters(
        content_type="np", foo="foo_1", bar="'bar_1"
    ),
)

# Request 2
types.ResponseOutput(
    name="foo",
    datatype="INT32",
    shape=[1, 3],
    data=[4, 5, 6],
    parameters=types.Parameters(
        content_type="np", foo="foo_2", bar="bar_2"
    ),
)

# Request 3
types.ResponseOutput(
    name="foo",
    datatype="INT32",
    shape=[1, 3],
```

(continues on next page)

(continued from previous page)

```
    data=[7, 8, 9],
    parameters=types.Parameters(content_type="np", bar="bar_3"),
)
```

CHAPTER

FIVE

CUSTOM INFERENCE RUNTIMES

There may be cases where the *inference runtimes* offered out-of-the-box by MLServer may not be enough, or where you may need **extra custom functionality** which is not included in MLServer (e.g. custom codecs). To cover these cases, MLServer lets you create custom runtimes very easily.

This page covers some of the bigger points that need to be taken into account when extending MLServer. You can also see this *end-to-end example* which walks through the process of writing a custom runtime.

5.1 Writing a custom inference runtime

MLServer is designed as an easy-to-extend framework, encouraging users to write their own custom runtimes easily. The starting point for this is the `MLModel` abstract class, whose main methods are:

- `load()`: Responsible for loading any artifacts related to a model (e.g. model weights, pickle files, etc.).
- `predict()`: Responsible for using a model to perform inference on an incoming data point.

Therefore, the *"one-line version"* of how to write a custom runtime is to write a custom class extending from `MLModel`, and then overriding those methods with your custom logic.

```
from mlserver import MLModel
from mlserver.types import InferenceRequest, InferenceResponse

class MyCustomRuntime(MLModel):

  async def load(self) -> bool:
    # TODO: Replace for custom logic to load a model artifact
    self._model = load_my_custom_model()
    return True

  async def predict(self, payload: InferenceRequest) -> InferenceResponse:
    # TODO: Replace for custom logic to run inference
    return self._model.predict(payload)
```

5.1.1 Simplified interface

MLServer exposes an alternative *"simplified" interface* which can be used to write custom runtimes. This interface can be enabled by decorating your `predict()` method with the `mlserver.codecs.decode_args` decorator. This will let you specify in the method signature both how you want your request payload to be decoded and how to encode the response back.

Based on the information provided in the method signature, MLServer will automatically decode the request payload into the different inputs specified as keyword arguments. Under the hood, this is implemented through *MLServer's codecs and content types system*.

Note: MLServer's *"simplified" interface* aims to cover use cases where encoding / decoding can be done through one of the codecs built-in into the MLServer package. However, there are instances where this may not be enough (e.g. variable number of inputs, variable content types, etc.). For these types of cases, please use MLServer's "advanced" interface, where you will have full control over the full encoding / decoding process.

As an example of the above, let's assume a model which

- Takes two lists of strings as inputs:
  - `questions`, containing multiple questions to ask our model.
  - `context`, containing multiple contexts for each of the questions.
- Returns a Numpy array with some predictions as the output.

Leveraging MLServer's simplified notation, we can represent the above as the following custom runtime:

```
from mlserver import MLModel
from mlserver.codecs import decode_args

class MyCustomRuntime(MLModel):

  async def load(self) -> bool:
    # TODO: Replace for custom logic to load a model artifact
    self._model = load_my_custom_model()
    return True

  @decode_args
  async def predict(self, questions: List[str], context: List[str]) -> np.ndarray:
    # TODO: Replace for custom logic to run inference
    return self._model.predict(questions, context)
```

Note that, the method signature of our `predict` method now specifies:

- The input names that we should be looking for in the request payload (i.e. `questions` and `context`).
- The expected content type for each of the request inputs (i.e. `List[str]` on both cases).
- The expected content type of the response outputs (i.e. `np.ndarray`).

5.1.2 Read and write headers

Note: The `headers` field within the `parameters` section of the request / response is managed by MLServer. Therefore, incoming payloads where this field has been explicitly modified will be overriden.

There are occasions where custom logic must be made conditional to extra information sent by the client outside of the payload. To allow for these use cases, MLServer will map all incoming HTTP headers (in the case of REST) or metadata (in the case of gRPC) into the `headers` field of the `parameters` object within the `InferenceRequest` instance.

```
from mlserver import MLModel
from mlserver.types import InferenceRequest, InferenceResponse

class CustomHeadersRuntime(MLModel):

  ...

  async def predict(self, payload: InferenceRequest) -> InferenceResponse:
    if payload.parameters and payload.parametes.headers:
      # These are all the incoming HTTP headers / gRPC metadata
      print(payload.parameters.headers)
    ...
```

Similarly, to return any HTTP headers (in the case of REST) or metadata (in the case of gRPC), you can append any values to the `headers` field within the `parameters` object of the returned `InferenceResponse` instance.

```
from mlserver import MLModel
from mlserver.types import InferenceRequest, InferenceResponse

class CustomHeadersRuntime(MLModel):

  ...

  async def predict(self, payload: InferenceRequest) -> InferenceResponse:
    ...
    return InferenceResponse(
      # Include any actual outputs from inference
      outputs=[],
      parameters=Parameters(headers={"foo": "bar"})
    )
```

5.2 Loading a custom MLServer runtime

MLServer lets you load custom runtimes dynamically into a running instance of MLServer. Once you have your custom runtime ready, all you need to is to move it to your model folder, next to your `model-settings.json` configuration file.

For example, if we assume a flat model repository where each folder represents a model, you would end up with a folder structure like the one below:

```
.
└── models
    └── sum-model
        ├── model-settings.json
        ├── models.py
```

Note that, from the example above, we are assuming that:

- Your custom runtime code lives in the `models.py` file.
- The `implementation` field of your `model-settings.json` configuration file contains the import path of your custom runtime (e.g. `models.MyCustomRuntime`).

```
{
  "model": "sum-model",
  "implementation": "models.MyCustomRuntime"
}
```

5.2.1 Loading a custom Python environment

More often that not, your custom runtimes will depend on external 3rd party dependencies which are not included within the main MLServer package. In these cases, to load your custom runtime, MLServer will need access to these dependencies.

It is possible to load this custom set of dependencies by providing them through an *environment tarball*, whose path can be specified within your `model-settings.json` file.

Warning: To load a custom environment, *parallel inference* **must** be enabled.

Warning: When loading custom environments, MLServer will always use the same Python interpreter that is used to run the main process. In other words, all custom environments will use the same version of Python than the main MLServer process.

If we take the *previous example* above as a reference, we could extend it to include our custom environment as:

```
.
└── models
    └── sum-model
        ├── environment.tar.gz
        ├── model-settings.json
        ├── models.py
```

Note that, in the folder layout above, we are assuming that:

- The `environment.tar.gz` tarball contains a pre-packaged version of your custom environment.
- The `environment_tarball` field of your `model-settings.json` configuration file points to your pre-packaged custom environment (i.e. `./environment.tar.gz`).

```
{
  "model": "sum-model",
```

(continues on next page)

(continued from previous page)

```
  "implementation": "models.MyCustomRuntime",
  "parameters": {
    "environment_tarball": "./environment.tar.gz"
  }
}
```

5.3 Building a custom MLServer image

Note: The `mlserver build` command expects that a Docker runtime is available and running in the background.

MLServer offers built-in utilities to help you build a custom MLServer image. This image can contain any custom code (including custom inference runtimes), as well as any custom environment, provided either through a Conda environment file or a `requirements.txt` file.

To leverage these, we can use the `mlserver build` command. Assuming that we're currently on the folder containing our custom inference runtime, we should be able to just run:

```
mlserver build . -t my-custom-server
```

The output will be a Docker image named `my-custom-server`, ready to be used.

5.3.1 Custom Environment

The *`mlserver build`* subcommand will search for any Conda environment file (i.e. named either as `environment.yaml` or `conda.yaml`) and / or any `requirements.txt` present in your root folder. These can be used to tell MLServer what Python environment is required in the final Docker image.

Note: The environment built by the `mlserver build` will be global to the whole MLServer image (i.e. every loaded model will, by default, use that custom environment). For Multi-Model Serving scenarios, it may be better to use *per-model custom environments* instead - which will allow you to run multiple custom environments at the same time.

5.3.2 Default Settings

The `mlserver build` subcommand will treat any *`settings.json`* or *`model-settings.json`* files present on your root folder as the default settings that must be set in your final image. Therefore, these files can be used to configure things like the default inference runtime to be used, or to even include **embedded models** that will always be present within your custom image.

Note: Default setting values can still be overriden by external environment variables or model-specific `model-settings.json`.

5.3.3 Custom Dockerfile

Out-of-the-box, the `mlserver build` subcommand leverages a default `Dockerfile` which takes into account a number of requirements, like

- Supporting arbitrary user IDs.
- Building your *base custom environment* on the fly.
- Configure a set of *default setting values*.

However, there may be occasions where you need to customise your `Dockerfile` even further. This may be the case, for example, when you need to provide extra environment variables or when you need to customise your Docker build process (e.g. by using other *"Docker-less"* tools, like Kaniko or Buildah).

To account for these cases, MLServer also includes a *`mlserver dockerfile`* subcommand which will just generate a `Dockerfile` (and optionally a `.dockerignore` file) exactly like the one used by the `mlserver build` command. This `Dockerfile` can then be customised according to your needs.

Note: The base `Dockerfile` requires Docker's Buildkit to be enabled. To ensure BuildKit is used, you can use the `DOCKER_BUILDKIT=1` environment variable, e.g.

```
DOCKER_BUILDKIT=1 docker build . -t my-custom-runtime:0.1.0
```

CHAPTER

SIX

METRICS

Out-of-the-box, MLServer exposes a set of metrics that help you monitor your machine learning workloads in production. These include standard metrics like number of requests and latency.

On top of these, you can also register and track your own *custom metrics* as part of your *custom inference runtimes*.

6.1 Default Metrics

By default, MLServer will expose metrics around inference requests (count and error rate) and the status of its internal requests queues. These internal queues are used for *adaptive batching* and *communication with the inference workers*.

Metric Name	Description
`model_infer_request_success`	Number of successful inference requests.
`model_infer_request_failure`	Number of failed inference requests.
`batch_request_queue`	Queue size for the *adaptive batching* queue.
`parallel_request_queue`	Queue size for the *inference workers* queue.

6.1.1 REST Server Metrics

On top of the default set of metrics, MLServer's REST server will also expose a set of metrics specific to REST.

Note: The prefix for the REST-specific metrics will be dependent on the `metrics_rest_server_prefix` flag from the *MLServer settings*.

Metric Name	Description
`[rest_server]_requests`	Number of REST requests, labelled by endpoint and status code.
`[rest_server]_requests_duration_seconds`	Latency of REST requests.
`[rest_server]_requests_in_progress`	Number of in-flight REST requests.

6.1.2 gRPC Server Metrics

On top of the default set of metrics, MLServer's gRPC server will also expose a set of metrics specific to gRPC.

Metric Name	Description
`grpc_server_handled`	Number of gRPC requests, labelled by gRPC code and method.
`grpc_server_started`	Number of in-flight gRPC requests.

6.2 Custom Metrics

MLServer allows you to register custom metrics within your custom inference runtimes. This can be done through the `mlserver.register()` and `mlserver.log()` methods.

- *`mlserver.register()`*: Register a new metric.
- *`mlserver.log()`*: Log a new set of metric / value pairs. If there's any unregistered metric, it will get registered on-the-fly.

Note: Under the hood, metrics logged through the *`mlserver.log()`* method will get exposed to Prometheus as a Histogram.

Custom metrics will generally be registered in the *`load()`* method and then used in the *`predict()`* method of your *custom runtime*.

```
import mlserver

from mlserver.types import InferenceRequest, InferenceResponse

class MyCustomRuntime(mlserver.MLModel):
  async def load(self) -> bool:
    self._model = load_my_custom_model()
    mlserver.register("my_custom_metric", "This is a custom metric example")
    return True

  async def predict(self, payload: InferenceRequest) -> InferenceResponse:
    mlserver.log(my_custom_metric=34)
    # TODO: Replace for custom logic to run inference
    return self._model.predict(payload)
```

6.3 Metrics Labelling

For metrics specific to a model (e.g. *custom metrics*, request counts, etc), MLServer will always label these with the **model name** and **model version**. Downstream, this will allow to aggregate and query metrics per model.

Note: If these labels are not present on a specific metric, this means that those metrics can't be sliced at the model level.

Below, you can find the list of standardised labels that you will be able to find on model-specific metrics:

Label Name	Description
`model_name`	Model Name (e.g. `my-custom-model`)
`model_version`	Model Version (e.g. `v1.2.3`)

6.4 Settings

MLServer will expose metric values through a metrics endpoint exposed on its own metric server. This endpoint can be polled by Prometheus or other OpenMetrics-compatible backends.

Below you can find the *settings* available to control the behaviour of the metrics server:

Label Name	Description	Default
`metrics_endp`	Path under which the metrics endpoint will be exposed.	`/metrics`
`metrics_port`	Port used to serve the metrics server.	`8082`
`metrics_rest`	Prefix used for metric names specific to MLServer's REST inference interface.	`rest_server`
`metrics_dir`	Directory used to store internal metric files (used to support metrics sharing across *inference workers*). This is equivalent to Prometheus' `$PROMETHEUS_MULTIPROC_DIR` env var.	MLServer's current working directory (i.e. `$PWD`)

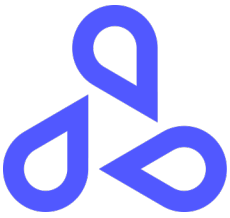
CHAPTER

SEVEN

DEPLOYMENT

MLServer is currently used as the core Python inference server in some of most popular Kubernetes-native serving frameworks, including Seldon Core and KServe (formerly known as KFServing). This allows MLServer users to leverage the usability and maturity of these frameworks to take their model deployments to the next level of their MLOps journey, ensuring that they are served in a robust and scalable infrastructure.

Note: In general, it should be possible to deploy models using MLServer into **any serving engine compatible with the V2 protocol**. Alternatively, it's also possible to manage MLServer deployments manually as regular processes (i.e. in a non-Kubernetes-native way). However, this may be more involved and highly dependant on the deployment infrastructure.



Deploy with Seldon Core

Deploy with KServe

7.1 Deployment with Seldon Core

MLServer is used as the core Python inference server in Seldon Core. Therefore, it should be straightforward to deploy your models either by using one of the built-in pre-packaged servers or by pointing to a *custom image of MLServer*.

Note: This section assumes a basic knowledge of Seldon Core and Kubernetes, as well as access to a working Kubernetes cluster with Seldon Core installed. To learn more about Seldon Core or how to install it, please visit the Seldon Core documentation.

7.1.1 Pre-packaged Servers

Out of the box, Seldon Core comes a few MLServer runtimes pre-configured to run straight away. This allows you to deploy a MLServer instance by just pointing to where your model artifact is and specifying what ML framework was used to train it.

Usage

To let Seldon Core know what framework was used to train your model, you can use the `implementation` field of your `SeldonDeployment` manifest. For example, to deploy a Scikit-Learn artifact stored remotely in GCS, one could do:

```
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: my-model
spec:
  protocol: v2
  predictors:
    - name: default
      graph:
        name: classifier
        implementation: SKLEARN_SERVER
        modelUri: gs://seldon-models/sklearn/iris
```

As you can see highlighted above, all that we need to specify is that:

- Our **inference deployment should use the V2 inference protocol**, which is done by **setting the `protocol` field to `kfserving`**.
- Our **model artifact is a serialised Scikit-Learn model**, therefore it should be served using the *MLServer SKLearn runtime*, which is done by **setting the `implementation` field to `SKLEARN_SERVER`**.

Note that, while the `protocol` should always be set to `kfserving` (i.e. so that models are served using the V2 inference protocol), the value of the `implementation` field will be dependant on your ML framework. The valid values of the `implementation` field are pre-determined by Seldon Core. However, it should also be possible to configure and add new ones (e.g. to support a *custom MLServer runtime*).

Once you have your `SeldonDeployment` manifest ready, then the next step is to apply it to your cluster. There are multiple ways to do this, but the simplest is probably to just apply it directly through `kubectl`, by running:

```
kubectl apply -f my-seldondeployment-manifest.yaml
```

To consult the supported values of the `implementation` field where MLServer is used, you can check the support table below.

Supported Pre-packaged Servers

As mentioned above, pre-packaged servers come built-in into Seldon Core. Therefore, only a pre-determined subset of them will be supported for a given release of Seldon Core.

The table below shows a list of the currently supported values of the `implementation` field. Each row will also show what ML framework they correspond to and also what MLServer runtime will be enabled internally on your model deployment when used.

Framework	MLServer Runtime	Seldon Core Pre-packaged Server	Documentation
Scikit-Learn	*MLServer SKLearn*	`SKLEARN_SERVER`	SKLearn Server
XGBoost	*MLServer XGBoost*	`XGBOOST_SERVER`	XGBoost Server
MLflow	*MLServer MLflow*	`MLFLOW_SERVER`	MLflow Server
Tempo	Tempo	`TEMPO_SERVER`	Tempo Server

Note that, on top of the ones shown above (backed by MLServer), Seldon Core **also provides a wider set** of pre-packaged servers. To check the full list, please visit the Seldon Core documentation.

7.1.2 Custom Runtimes

There could be cases where the pre-packaged MLServer runtimes supported out-of-the-box in Seldon Core may not be enough for our use case. The framework provided by MLServer makes it easy to *write custom runtimes*, which can then get packaged up as images. These images then become self-contained model servers with your custom runtime. Therefore Seldon Core makes it as easy to deploy them into your serving infrastructure.

Usage

The `componentSpecs` field of the `SeldonDeployment` manifest will allow us to let Seldon Core know what image should be used to serve a custom model. For example, if we assume that our custom image has been tagged as `my-custom-server:0.1.0`, we could write our `SeldonDeployment` manifest as follows:

```
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: my-model
spec:
  protocol: v2
  predictors:
    - name: default
      graph:
        name: classifier
      componentSpecs:
        - spec:
            containers:
              - name: classifier
                image: my-custom-server:0.1.0
```

As we can see highlighted on the snippet above, all that's needed to deploy a custom MLServer image is:

- Letting Seldon Core know that the model deployment will be served through the V2 inference protocol) by setting the `protocol` field to `v2`.

- Pointing our model container to use our **custom MLServer image**, by specifying it on the `image` field of the `componentSpecs` section of the manifest.

Once you have your `SeldonDeployment` manifest ready, then the next step is to apply it to your cluster. There are multiple ways to do this, but the simplest is probably to just apply it directly through `kubectl`, by running:

```
kubectl apply -f my-seldondeployment-manifest.yaml
```

7.2 Deployment with KServe

MLServer is used as the core Python inference server in KServe (formerly known as KFServing). This allows for a straightforward avenue to deploy your models into a scalable serving infrastructure backed by Kubernetes.

Note: This section assumes a basic knowledge of KServe and Kubernetes, as well as access to a working Kubernetes cluster with KServe installed. To learn more about KServe or how to install it, please visit the KServe documentation.

7.2.1 Serving Runtimes

KServe provides built-in serving runtimes to deploy models trained in common ML frameworks. These allow you to deploy your models into a robust infrastructure by just pointing to where the model artifacts are stored remotely.

Some of these runtimes leverage MLServer as the core inference server. Therefore, it should be straightforward to move from your local testing to your serving infrastructure.

Usage

To use any of the built-in serving runtimes offered by KServe, it should be enough to select the relevant one your `InferenceService` manifest.

For example, to serve a Scikit-Learn model, you could use a manifest like the one below:

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: my-model
spec:
  predictor:
    sklearn:
      protocolVersion: v2
      storageUri: gs://seldon-models/sklearn/iris
```

As you can see highlighted above, the `InferenceService` manifest will only need to specify the following points:

- The model artifact is a Scikit-Learn model. Therefore, we will use the `sklearn` serving runtime to deploy it.
- The model will be served using the V2 inference protocol, which can be enabled by setting the `protocolVersion` field to `v2`.

Once you have your `InferenceService` manifest ready, then the next step is to apply it to your cluster. There are multiple ways to do this, but the simplest is probably to just apply it directly through `kubectl`, by running:

```
kubectl apply -f my-inferenceservice-manifest.yaml
```

Supported Serving Runtimes

As mentioned above, KServe offers support for built-in serving runtimes, some of which leverage MLServer as the inference server. Below you can find a table listing these runtimes, and the MLServer inference runtime that they correspond to.

Framework	MLServer Runtime	KServe Serving Runtime	Documentation
Scikit-Learn	*MLServer SKLearn*	`sklearn`	SKLearn Serving Runtime
XGBoost	*MLServer XGBoost*	`xgboost`	XGBoost Serving Runtime

Note that, on top of the ones shown above (backed by MLServer), KServe also provides a wider set of serving runtimes. To see the full list, please visit the KServe documentation.

7.2.2 Custom Runtimes

Sometimes, the serving runtimes built into KServe may not be enough for our use case. The framework provided by MLServer makes it easy to *write custom runtimes*, which can then get packaged up as images. These images then become self-contained model servers with your custom runtime. Therefore, it's easy to deploy them into your serving infrastructure leveraging KServe support for custom runtimes.

Usage

The `InferenceService` manifest gives you full control over the containers used to deploy your machine learning model. This can be leveraged to point your deployment to the *custom MLServer image containing your custom logic*. For example, if we assume that our custom image has been tagged as `my-custom-server:0.1.0`, we could write an `InferenceService` manifest like the one below:

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: my-model
spec:
  predictor:
    containers:
      - name: classifier
        image: my-custom-server:0.1.0
        env:
          - name: PROTOCOL
            value: v2
        ports:
          - containerPort: 8080
            protocol: TCP
```

As we can see highlighted above, the main points that we'll need to take into account are:

- Pointing to our custom MLServer `image` in the custom container section of our `InferenceService`.
- Explicitly choosing the V2 inference protocol to serve our model.

- Let KServe know what port will be exposed by our custom container to send inference requests.

Once you have your `InferenceService` manifest ready, then the next step is to apply it to your cluster. There are multiple ways to do this, but the simplest is probably to just apply it directly through `kubectl`, by running:

```
kubectl apply -f my-inferenceservice-manifest.yaml
```

CHAPTER

EIGHT

INFERENCE RUNTIMES

Inference runtimes allow you to define how your model should be used within MLServer. You can think of them as the **backend glue** between MLServer and your machine learning framework of choice.

Out of the box, MLServer comes with a set of pre-packaged runtimes which let you interact with a subset of common ML frameworks. This allows you to start serving models saved in these frameworks straight away. To avoid bringing in dependencies for frameworks that you don't need to use, these runtimes are implemented as independent (and optional) Python packages. This mechanism also allows you to **rollout your *own custom runtimes* very easily**.

To pick which runtime you want to use for your model, you just need to make sure that the right package is installed, and then point to the correct runtime class in your `model-settings.json` file.

8.1 Included Inference Runtimes

Framework	Package Name	Implementation Class	Example	Documentation
Scikit-Learn	`mlserver-sklearn`	`mlserver_sklearn.SKLearnModel`	*Scikit-Learn example*	*MLServer SKLearn*
XGBoost	`mlserver-xgboost`	`mlserver_xgboost.XGBoostModel`	*XGBoost example*	*MLServer XGBoost*
HuggingFace	`mlserver-hugging`	`mlserver_huggingface.HuggingFaceRuntime`	*HuggingFace example*	*MLServer HuggingFace*
Spark MLlib	`mlserver-mllib`	`mlserver_mllib.MLlibModel`	Coming Soon	*MLServer MLlib*
LightGBM	`mlserver-lightgb`	`mlserver_lightgbm.LightGBMModel`	*LightGBM example*	*MLServer LightGBM*
Tempo	`tempo`	`tempo.mlserver.InferenceRuntime`	*Tempo example*	`github.com/SeldonIO/tempo`
MLflow	`mlserver-mlflow`	`mlserver_mlflow.MLflowRuntime`	*MLflow example*	*MLServer MLflow*
Alibi-Detect	`mlserver-alibi-d`	`mlserver_alibi_detect.AlibiDetectRuntime`	*Alibi-detect example*	*MLServer Alibi-Detect*
Alibi-Explain	`mlserver-alibi-e`	`mlserver_alibi_explain.AlibiExplainRuntime`	Coming Soon	*MLServer Alibi-Explain*

8.1.1 Scikit-Learn runtime for MLServer

This package provides a MLServer runtime compatible with Scikit-Learn.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-sklearn
```

For further information on how to use MLServer with Scikit-Learn, you can check out this *worked out example*.

Content Types

If no *content type* is present on the request or metadata, the Scikit-Learn runtime will try to decode the payload as a *NumPy Array*. To avoid this, either send a different content type explicitly, or define the correct one as part of your *model's metadata*.

Model Outputs

The Scikit-Learn inference runtime exposes a number of outputs depending on the model type. These outputs match to the `predict`, `predict_proba` and `transform` methods of the Scikit-Learn model.

Output	Returned By Default	Availability
`predict`		Available on most models, but not in Scikit-Learn pipelines.
`predict_proba`		Only available on non-regressor models.
`transform`		Only availabe on Scikit-Learn pipelines.

By default, the runtime will only return the output of `predict`. However, you are able to control which outputs you want back through the `outputs` field of your *`InferenceRequest`* payload.

For example, to only return the model's `predict_proba` output, you could define a payload such as:

```
{
  "inputs": [
    {
      "name": "my-input",
      "datatype": "INT32",
      "shape": [2, 2],
      "data": [1, 2, 3, 4]
    }
  ],
  "outputs": [
    { "name": "predict_proba" }
  ]
}
```

8.1.2 XGBoost runtime for MLServer

This package provides a MLServer runtime compatible with XGBoost.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-xgboost
```

For further information on how to use MLServer with XGBoost, you can check out this *worked out example*.

XGBoost Artifact Type

The XGBoost inference runtime will expect that your model is serialised via one of the following methods:

Extension	Docs	Example
`*.json`	JSON Format	`booster.save_model("model.json")`
`*.ubj`	Binary JSON Format	`booster.save_model("model.ubj")`
`*.bst`	(Old) Binary Format	`booster.save_model("model.bst")`

Note: By default, the runtime will look for a file called `model.[json | ubj | bst]`. However, this can be modified through the `parameters.uri` field of your *`ModelSettings`* config (see the section on *Model Settings* for more details).

```
{
  "name": "foo",
  "parameters": {
    "uri": "./my-own-model-filename.json"
  }
}
```

Content Types

If no *content type* is present on the request or metadata, the XGBoost runtime will try to decode the payload as a *NumPy Array*. To avoid this, either send a different content type explicitly, or define the correct one as part of your *model's metadata*.

Model Outputs

The XGBoost inference runtime exposes a number of outputs depending on the model type. These outputs match to the `predict` and `predict_proba` methods of the XGBoost model.

Output	Returned By Default	Availability
`predict`		Available on all XGBoost models.
`predict_proba`		Only available on non-regressor models (i.e. `XGBClassifier` models).

By default, the runtime will only return the output of `predict`. However, you are able to control which outputs you want back through the `outputs` field of your *InferenceRequest* payload.

For example, to only return the model's `predict_proba` output, you could define a payload such as:

```
{
  "inputs": [
    {
      "name": "my-input",
      "datatype": "INT32",
      "shape": [2, 2],
      "data": [1, 2, 3, 4]
    }
  ],
  "outputs": [
    { "name": "predict_proba" }
  ]
}
```

8.1.3 MLflow runtime for MLServer

This package provides a MLServer runtime compatible with MLflow models.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-mlflow
```

Content Types

The MLflow inference runtime introduces a new `dict` content type, which decodes an incoming V2 request as a dictionary of tensors. This is useful for certain MLflow-serialised models, which will expect that the model inputs are serialised in this format.

Note: The `dict` content type can be *stacked* with other content types, like *np*. This allows the user to use a different set of content types to decode each of the dict entries.

8.1.4 Spark MLlib runtime for MLServer

This package provides a MLServer runtime compatible with Spark MLlib.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-mllib
```

For further information on how to use MLServer with Spark MLlib, you can check out the MLServer repository.

8.1.5 LightGBM runtime for MLServer

This package provides a MLServer runtime compatible with LightGBM.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-lightgbm
```

For further information on how to use MLServer with LightGBM, you can check out this *worked out example*.

Content Types

If no *content type* is present on the request or metadata, the LightGBM runtime will try to decode the payload as a *NumPy Array*. To avoid this, either send a different content type explicitly, or define the correct one as part of your *model's metadata*.

8.1.6 Alibi-Detect runtime for MLServer

This package provides a MLServer runtime compatible with alibi-detect models.

Usage

You can install the `mlserver-alibi-detect` runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-alibi-detect
```

For further information on how to use MLServer with Alibi-Detect, you can check out this *worked out example*.

Content Types

If no *content type* is present on the request or metadata, the Alibi-Detect runtime will try to decode the payload as a *NumPy Array*. To avoid this, either send a different content type explicitly, or define the correct one as part of your *model's metadata*.

Settings

The Alibi Detect runtime exposes a couple setting flags which can be used to customise how the runtime behaves. These settings can be added under the `parameters.extra` section of your `model-settings.json` file, e.g.

```
{
  "name": "drift-detector",
  "implementation": "mlserver_alibi_detect.AlibiDetectRuntime",
  "parameters": {
    "uri": "./alibi-detect-artifact/",
    "extra": {
      "batch_size": 5
    }
  }
}
```

Reference

You can find the full reference of the accepted extra settings for the Alibi Detect runtime below:

pydantic settings mlserver_alibi_detect.runtime.**AlibiDetectSettings**

Parameters that apply only to alibi detect models

Config

- **env_prefix**: *str* = *MLSERVER_MODEL_ALIBI_DETECT_*

Fields

- *batch_size (int | None)*
- *predict_parameters (dict)*
- *state_save_freq (int | None)*

field batch_size: int | None = None

Run the detector after accumulating a batch of size *batch_size*. Note that this is different to MLServer's adaptive batching, since the rest of requests will just return empty (i.e. instead of being hold until inference runs for all of them).

field predict_parameters: dict = {}

Keyword parameters to pass to the detector's *predict* method.

field state_save_freq: int | None = 100

Save the detector state after every *state_save_freq* predictions. Only applicable to detectors with a *save_state* method.

Constraints

- **exclusiveMinimum** = 0

8.1.7 Alibi-Explain runtime for MLServer

This package provides a MLServer runtime compatible with Alibi-Explain.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-alibi-explain
```

8.1.8 HuggingFace runtime for MLServer

This package provides a MLServer runtime compatible with HuggingFace Transformers.

Usage

You can install the runtime, alongside `mlserver`, as:

```
pip install mlserver mlserver-huggingface
```

For further information on how to use MLServer with HuggingFace, you can check out this *worked out example*.

Content Types

The HuggingFace runtime will always decode the input request using its own built-in codec. Therefore, *content type annotations* at the request level will **be ignored**. Not that this **doesn't include *input-level content type* annotations**, which will be respected as usual.

Settings

The HuggingFace runtime exposes a couple extra parameters which can be used to customise how the runtime behaves. These settings can be added under the `parameters.extra` section of your `model-settings.json` file, e.g.

```
{
  "name": "qa",
  "implementation": "mlserver_huggingface.HuggingFaceRuntime",
  "parameters": {
    "extra": {
      "task": "question-answering",
      "optimum_model": true
    }
  }
}
```

Note: These settings can also be injected through environment variables prefixed with `MLSERVER_MODEL_HUGGINGFACE_`, e.g.

```
MLSERVER_MODEL_HUGGINGFACE_TASK="question-answering"
MLSERVER_MODEL_HUGGINGFACE_OPTIMUM_MODEL=true
```

Reference

You can find the full reference of the accepted extra settings for the HuggingFace runtime below:

pydantic settings mlserver_huggingface.settings.**HuggingFaceSettings**

Parameters that apply only to HuggingFace models

Config

- **env_prefix**: *str* = *MLSERVER_MODEL_HUGGINGFACE_*

Fields

- *device (int)*
- *framework (str | None)*
- *inter_op_threads (int | None)*
- *intra_op_threads (int | None)*
- *optimum_model (bool)*
- *pretrained_model (str | None)*
- *pretrained_tokenizer (str | None)*
- *task (str)*
- *task_suffix (str)*

field device: int = -1

Device in which this pipeline will be loaded (e.g., "cpu", "cuda:1", "mps", or a GPU ordinal rank like 1).

field framework: str | None = None

The framework to use, either "pt" for PyTorch or "tf" for TensorFlow.

field inter_op_threads: int | None = None

Threads used for parallelism between independent operations. PyTorch: https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html Tensorflow: https://www.tensorflow.org/api_docs/python/tf/config/threading/set_inter_op_parallelism_threads

field intra_op_threads: int | None = None

Threads used within an individual op for parallelism. PyTorch: https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html Tensorflow: https://www.tensorflow.org/api_docs/python/tf/config/threading/set_intra_op_parallelism_threads

field optimum_model: bool = False

Flag to decide whether the pipeline should use a Optimum-optimised model or the standard Transformers model. Under the hood, this will enable the model to use the optimised ONNX runtime.

field pretrained_model: str | None = None

Name of the model that should be loaded in the pipeline.

field pretrained_tokenizer: str | None = None

Name of the tokenizer that should be loaded in the pipeline.

field task: str = ''

Pipeline task to load. You can see the available Optimum and Transformers tasks available in the links below:

- Optimum Tasks

- Transformer Tasks

field task_suffix: str = ''

Suffix to append to the base task name. Useful for, e.g. translation tasks which require a suffix on the task name to specify source and target.

property task_name

8.1.9 Custom Inference Runtimes

There may be cases where the *inference runtimes* offered out-of-the-box by MLServer may not be enough, or where you may need **extra custom functionality** which is not included in MLServer (e.g. custom codecs). To cover these cases, MLServer lets you create custom runtimes very easily.

To learn more about how you can write custom runtimes with MLServer, check out the *Custom Runtimes user guide*. Alternatively, you can also see this *end-to-end example* which walks through the process of writing a custom runtime.

CHAPTER

NINE

REFERENCE

9.1 MLServer Settings

MLServer can be configured through a `settings.json` file on the root folder from where MLServer is started. Note that these are server-wide settings (e.g. gRPC or HTTP port) which are separate from the *invidual model settings*. Alternatively, this configuration can also be passed through **environment variables** prefixed with `MLSERVER_` (e.g. `MLSERVER_GRPC_PORT`).

9.1.1 Settings

pydantic settings mlserver.settings.**Settings**

Config

- **env_file**: *str* = *.env*
- **env_prefix**: *str* = *MLSERVER_*

Fields

- *cors_settings (mlserver.settings.CORSSettings | None)*
- *debug (bool)*
- *environments_dir (str)*
- *extensions (List[str])*
- *grpc_max_message_length (int | None)*
- *grpc_port (int)*
- *host (str)*
- *http_port (int)*
- *kafka_enabled (bool)*
- *kafka_servers (str)*
- *kafka_topic_input (str)*
- *kafka_topic_output (str)*
- *load_models_at_startup (bool)*
- *logging_settings (str | Dict | None)*
- *metrics_dir (str)*

- *metrics_endpoint (str | None)*
- *metrics_port (int)*
- *metrics_rest_server_prefix (str)*
- *model_repository_implementation (pydantic.types.PyObject | None)*
- *model_repository_implementation_args (dict)*
- *model_repository_root (str)*
- *parallel_workers (int)*
- *parallel_workers_timeout (int)*
- *root_path (str)*
- *server_name (str)*
- *server_version (str)*

field cors_settings: CORSSettings | None = None

field debug: bool = True

field environments_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/mlserver/checkouts/stable/docs/.envs'

Directory used to store custom environments. By default, the *.envs* folder of the current working directory will be used.

field extensions: List[str] = []

Server extensions loaded.

field grpc_max_message_length: int | None = None

Maximum length (i.e. size) of gRPC payloads.

field grpc_port: int = 8081

Port where to listen for gRPC connections.

field host: str = '0.0.0.0'

Host where to listen for connections.

field http_port: int = 8080

Port where to listen for HTTP / REST connections.

field kafka_enabled: bool = False

field kafka_servers: str = 'localhost:9092'

field kafka_topic_input: str = 'mlserver-input'

field kafka_topic_output: str = 'mlserver-output'

field load_models_at_startup: bool = True

Flag to load all available models automatically at startup.

field logging_settings: str | Dict | None = None

Path to logging config file or dictionary configuration.

field metrics_dir: str = '/home/docs/checkouts/readthedocs.org/user_builds/mlserver/checkouts/stable/docs/.metrics'

Directory used to share metrics across parallel workers. Equivalent to the *PROMETHEUS_MULTIPROC_DIR* env var in *prometheus-client*. Note that this won't be used if the *parallel_workers* flag is disabled. By default, the *.metrics* folder of the current working directory will be used.

field metrics_endpoint: str | None = '/metrics'

Endpoint used to expose Prometheus metrics. Alternatively, can be set to *None* to disable it.

field metrics_port: int = 8082

Port used to expose metrics endpoint.

field metrics_rest_server_prefix: str = 'rest_server'

Metrics rest server string prefix to be exported.

field model_repository_implementation: PyObject | None = None

Python path to the inference runtime to model repository (e.g. `mlserver.repository.repository.SchemalessModelRepository`).

field model_repository_implementation_args: dict = {}

Extra parameters for model repository.

field model_repository_root: str = '.'

Root of the model repository, where we will search for models.

field parallel_workers: int = 1

When parallel inference is enabled, number of workers to run inference across.

field parallel_workers_timeout: int = 5

Grace timeout to wait until the workers shut down when stopping MLServer.

field root_path: str = ''

Set the ASGI root_path for applications submounted below a given URL path.

field server_name: str = 'mlserver'

Name of the server.

field server_version: str = '1.3.5'

Version of the server.

9.2 Model Settings

In MLServer, each loaded model can be configured separately. This configuration will include model information (e.g. metadata about the accepted inputs), but also model-specific settings (e.g. number of *parallel workers* to run inference).

This configuration will usually be provided through a `model-settings.json` file which **sits next to the model artifacts**. However, it's also possible to provide this through environment variables prefixed with `MLSERVER_MODEL_` (e.g. `MLSERVER_MODEL_IMPLEMENTATION`). Note that, in the latter case, this environment variables will be shared across all loaded models (unless they get overriden by a `model-settings.json` file). Additionally, if no `model-settings.json` file is found, MLServer will also try to load a *"default"* model from these environment variables.

9.2.1 Settings

pydantic settings mlserver.settings.**ModelSettings**

Config

- **env_file**: *str* = *.env*
- **env_prefix**: *str* = *MLSERVER_MODEL_*
- **underscore_attrs_are_private**: *bool* = *True*

Fields

- *implementation_ (str)*
- *inputs (List[mlserver.types.dataplane.MetadataTensor])*
- *max_batch_size (int)*
- *max_batch_time (float)*
- *name (str)*
- *outputs (List[mlserver.types.dataplane.MetadataTensor])*
- *parallel_workers (int | None)*
- *parameters (mlserver.settings.ModelParameters | None)*
- *platform (str)*
- *versions (List[str])*
- *warm_workers (bool)*

field implementation_: str [Required] (alias 'implementation')

Python path to the inference runtime to use to serve this model (e.g. `mlserver_sklearn.SKLearnModel`).

field inputs: List[*MetadataTensor*] = []

Metadata about the inputs accepted by the model.

field max_batch_size: int = 0

When adaptive batching is enabled, maximum number of requests to group together in a single batch.

field max_batch_time: float = 0.0

When adaptive batching is enabled, maximum amount of time (in seconds) to wait for enough requests to build a full batch.

field name: str = ''

Name of the model.

field outputs: List[*MetadataTensor*] = []

Metadata about the outputs returned by the model.

field parallel_workers: int | None = None

Use the *parallel_workers* field the server wide settings instead.

field parameters: *ModelParameters* | None = None

Extra parameters for each instance of this model.

field platform: str = ''

Framework used to train and serialise the model (e.g. sklearn).

field versions: List[str] = []

Versions of dependencies used to train the model (e.g. sklearn/0.20.1).

field warm_workers: bool = False

Inference workers will now always be *warmed up* at start time.

classmethod parse_file(*path: str*) → *ModelSettings*

classmethod parse_obj(*obj: Any*) → *ModelSettings*

property implementation: Type[MLModel]

property version: str | None

9.2.2 Extra Model Parameters

pydantic settings mlserver.settings.**ModelParameters**

Parameters that apply only to a particular instance of a model. This can include things like model weights, or arbitrary `extra` parameters particular to the underlying inference runtime. The main difference with respect to `ModelSettings` is that parameters can change on each instance (e.g. each version) of the model.

Config

- **env_file**: *str = .env*
- **env_prefix**: *str = MLSERVER_MODEL_*
- **extra**: *Extra = allow*

Fields

- *content_type (str | None)*
- *environment_tarball (str | None)*
- *extra (dict | None)*
- *format (str | None)*
- *uri (str | None)*
- *version (str | None)*

field content_type: str | None = None

Default content type to use for requests and responses.

field environment_tarball: str | None = None

Path to the environment tarball which should be used to load this model.

field extra: dict | None = {}

Arbitrary settings, dependent on the inference runtime implementation.

field format: str | None = None

Format of the model (only available on certain runtimes).

field uri: str | None = None

URI where the model artifacts can be found. This path must be either absolute or relative to where MLServer is running.

field version: str | None = None

Version of the model.

9.3 MLServer CLI

The MLServer package includes a `mlserver` CLI designed to help with some of the common tasks involved with a model's lifecycle. Below, you can find the full list of supported subcommands. Note that you can also get a similar high-level outline at any time by running:

```
mlserver --help
```

9.3.1 Commands

mlserver

Command-line interface to manage MLServer models.

```
mlserver [OPTIONS] COMMAND [ARGS]...
```

Options

--version

Show the version and exit.

build

Build a Docker image for a custom MLServer runtime.

```
mlserver build [OPTIONS] FOLDER
```

Options

-t, --tag <tag>

--no-cache

Arguments

FOLDER

Required argument

dockerfile

Generate a Dockerfile

```
mlserver dockerfile [OPTIONS] FOLDER
```

Options

-i, --include-dockerignore

Arguments

FOLDER

Required argument

infer

Execute batch inference requests against V2 inference server (experimental).

```
mlserver infer [OPTIONS]
```

Options

-u, --url <url>

URL of the MLServer to send inference requests to. Should not contain http or https.

-m, --model-name <model_name>

Required Name of the model to send inference requests to.

-i, --input-data-path <input_data_path>

Required Local path to the input file containing inference requests to be processed.

-o, --output-data-path <output_data_path>

Required Local path to the output file for the inference responses to be written to.

-w, --workers <workers>

-r, --retries <retries>

-s, --batch-size <batch_size>

Send inference requests grouped together as micro-batches.

-b, --binary-data

Send inference requests as binary data (not fully supported).

-v, --verbose

Verbose mode.

-vv, --extra-verbose

Extra verbose mode (shows detailed requests and responses).

-t, --transport <transport>

Transport type to use to send inference requests. Can be 'rest' or 'grpc' (not yet supported).

Options

rest | grpc

-H, --request-headers <request_headers>

Headers to be set on each inference request send to the server. Multiple options are allowed as: -H 'Header1: Val1' -H 'Header2: Val2'. When setting up as environmental provide as 'Header1:Val1 Header2:Val2'.

--timeout <timeout>

Connection timeout to be passed to tritonclient.

--batch-interval <batch_interval>

Minimum time interval (in seconds) between requests made by each worker.

--batch-jitter <batch_jitter>

Maximum random jitter (in seconds) added to batch interval between requests.

--use-ssl

Use SSL in communications with inference server.

--insecure

Disable SSL verification in communications. Use with caution.

Environment variables

MLSERVER_INFER_URL

Provide a default for *--url*

MLSERVER_INFER_MODEL_NAME

Provide a default for *--model-name*

MLSERVER_INFER_INPUT_DATA_PATH

Provide a default for *--input-data-path*

MLSERVER_INFER_OUTPUT_DATA_PATH

Provide a default for *--output-data-path*

MLSERVER_INFER_WORKERS

Provide a default for *--workers*

MLSERVER_INFER_RETRIES

Provide a default for *--retries*

MLSERVER_INFER_BATCH_SIZE

Provide a default for *--batch-size*

MLSERVER_INFER_BINARY_DATA

Provide a default for *--binary-data*

MLSERVER_INFER_VERBOSE

Provide a default for *--verbose*

MLSERVER_INFER_EXTRA_VERBOSE

Provide a default for *--extra-verbose*

MLSERVER_INFER_TRANSPORT

Provide a default for *--transport*

MLSERVER_INFER_REQUEST_HEADERS

Provide a default for *--request-headers*

MLSERVER_INFER_CONNECTION_TIMEOUT

Provide a default for *--timeout*

MLSERVER_INFER_BATCH_INTERVAL

Provide a default for *--batch-interval*

MLSERVER_INFER_BATCH_JITTER

Provide a default for *--batch-jitter*

MLSERVER_INFER_USE_SSL

Provide a default for *--use-ssl*

MLSERVER_INFER_INSECURE

Provide a default for *--insecure*

init

Generate a base project template

```
mlserver init [OPTIONS]
```

Options

-t, --template <template>

start

Start serving a machine learning model with MLServer.

```
mlserver start [OPTIONS] FOLDER
```

Arguments

FOLDER
Required argument

9.4 Python API

MLServer can be installed as a Python package, which exposes a public framework which can be used to build *custom inference runtimes* and *codecs*.

Below, you can find the main reference for the Python API exposed by the MLServer framework.

9.4.1 MLModel

The `MLModel` class is the base class for all *custom inference runtimes*. It exposes the main interface that MLServer will use to interact with ML models.

The bulk of its public interface are the *`load()`* and *`predict()`* methods. However, it also contains helpers with encoding / decoding of requests and responses, as well as properties to access the most common bits of the model's metadata.

When writing *custom runtimes*, **this class should be extended to implement your own load and predict logic**.

class mlserver.**MLModel**(*settings:* ModelSettings)

Abstract inference runtime which exposes the main interface to interact with ML models.

async load() → bool

Method responsible for loading the model from a model artefact. This method will be called on each of the parallel workers (when *parallel inference*) is enabled). Its return value will represent the model's readiness status. A return value of `True` will mean the model is ready.

This method should be overriden to implement your custom load logic.

async predict(*payload:* InferenceRequest) → *InferenceResponse*

Method responsible for running inference on the model.

This method should be overriden to implement your custom inference logic.

property name: str

Model name, from the model settings.

property version: str | None

Model version, from the model settings.

property settings: *ModelSettings*

Model settings.

property inputs: List[*MetadataTensor*] | None

Expected model inputs, from the model settings.

Note that this property can also be modified at model's load time to inject any inputs metadata.

property outputs: List[*MetadataTensor*] | None

Expected model outputs, from the model settings.

Note that this property can also be modified at model's load time to inject any outputs metadata.

decode(*request_input:* RequestInput, *default_codec: Type[*InputCodec*]* | InputCodec | *None* = *None*) → Any

Helper to decode a **request input** into its corresponding high-level Python object. This method will find the most appropiate *input codec* based on the model's metadata and the input's content type. Otherwise, it will fall back to the codec specified in the `default_codec` kwarg.

decode_request(*inference_request:* InferenceRequest, *default_codec: Type[*RequestCodec*]* | RequestCodec | *None* = *None*) → Any

Helper to decode an **inference request** into its corresponding high-level Python object. This method will find the most appropiate *request codec* based on the model's metadata and the requests's content type. Otherwise, it will fall back to the codec specified in the `default_codec` kwarg.

encode_response(*payload: Any*, *default_codec: Type[*RequestCodec*]* | RequestCodec | *None* = *None*) → *InferenceResponse*

Helper to encode a high-level Python object into its corresponding **inference response**. This method will find the most appropiate *request codec* based on the payload's type. Otherwise, it will fall back to the codec specified in the `default_codec` kwarg.

encode(*payload: Any*, *request_output:* RequestOutput, *default_codec: Type[*InputCodec*]* | InputCodec | *None* = *None*) → *ResponseOutput*

Helper to encode a high-level Python object into its corresponding **response output**. This method will find the most appropiate *input codec* based on the model's metadata, request output's content type or payload's type. Otherwise, it will fall back to the codec specified in the `default_codec` kwarg.

9.4.2 Types

pydantic model mlserver.types.**InferenceErrorResponse**


```
{
   "title": "InferenceErrorResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "error": {
         "title": "Error",
         "type": "string"
      }
   }
}
```


Fields

- `error (str | None)`

field error: str | None = None

pydantic model mlserver.types.**InferenceRequest**


```
{
   "title": "InferenceRequest",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
```


(continues on next page)

(continued from previous page)


```
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "id": {
         "title": "Id",
         "type": "string"
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      },
      "inputs": {
         "title": "Inputs",
         "type": "array",
         "items": {
            "$ref": "#/definitions/RequestInput"
         }
      },
      "outputs": {
         "title": "Outputs",
         "type": "array",
         "items": {
            "$ref": "#/definitions/RequestOutput"
         }
      }
   },
   "required": [
      "inputs"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "content_type": {
               "title": "Content Type",
               "type": "string"
            },
            "headers": {
               "title": "Headers",
               "type": "object"
            }
         }
      },
      "TensorData": {
         "title": "TensorData",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525"
      },
```


(continues on next page)

(continued from previous page)


```
      "RequestInput": {
         "title": "RequestInput",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "shape": {
               "title": "Shape",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "datatype": {
               "title": "Datatype",
               "type": "string"
            },
            "parameters": {
               "$ref": "#/definitions/Parameters"
            },
            "data": {
               "$ref": "#/definitions/TensorData"
            }
         },
         "required": [
            "name",
            "shape",
            "datatype",
            "data"
         ]
      },
      "RequestOutput": {
         "title": "RequestOutput",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "parameters": {
               "$ref": "#/definitions/Parameters"
            }
         },
         "required": [
```


(continues on next page)

(continued from previous page)

```
            "name"
         ]
      }
   }
}
```

Fields

- id (str | None)
- inputs (List[mlserver.types.dataplane.RequestInput])
- outputs (List[mlserver.types.dataplane.RequestOutput] | None)
- parameters (mlserver.types.dataplane.Parameters | None)

field id: str | None = None

field inputs: List[*RequestInput*] [Required]

field outputs: List[*RequestOutput*] | None = None

field parameters: *Parameters* | None = None

pydantic model mlserver.types.**InferenceResponse**

```
{
   "title": "InferenceResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "model_name": {
         "title": "Model Name",
         "type": "string"
      },
      "model_version": {
         "title": "Model Version",
         "type": "string"
      },
      "id": {
         "title": "Id",
         "type": "string"
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      },
      "outputs": {
         "title": "Outputs",
         "type": "array",
         "items": {
            "$ref": "#/definitions/ResponseOutput"
         }
```

(continues on next page)

(continued from previous page)


```
      }
   },
   "required": [
      "model_name",
      "outputs"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "content_type": {
               "title": "Content Type",
               "type": "string"
            },
            "headers": {
               "title": "Headers",
               "type": "object"
            }
         }
      },
      "TensorData": {
         "title": "TensorData",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525"
      },
      "ResponseOutput": {
         "title": "ResponseOutput",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "shape": {
               "title": "Shape",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "datatype": {
               "title": "Datatype",
               "type": "string"
            },
```


(continues on next page)

(continued from previous page)


```
            "parameters": {
               "$ref": "#/definitions/Parameters"
            },
            "data": {
               "$ref": "#/definitions/TensorData"
            }
         },
         "required": [
            "name",
            "shape",
            "datatype",
            "data"
         ]
      }
   }
}
```


Fields

- id (str | None)
- model_name (str)
- model_version (str | None)
- outputs (List[mlserver.types.dataplane.ResponseOutput])
- parameters (mlserver.types.dataplane.Parameters | None)

field id: str | None = None

field model_name: str [Required]

field model_version: str | None = None

field outputs: List[*ResponseOutput*] [Required]

field parameters: *Parameters* | None = None

pydantic model mlserver.types.**MetadataModelErrorResponse**


```
{
   "title": "MetadataModelErrorResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "error": {
         "title": "Error",
         "type": "string"
      }
   },
   "required": [
      "error"
```


(continues on next page)

(continued from previous page)

```
    ]
}
```

Fields

- error (str)

field error: str [Required]

pydantic model mlserver.types.**MetadataModelResponse**


```
{
   "title": "MetadataModelResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "versions": {
         "title": "Versions",
         "type": "array",
         "items": {
            "type": "string"
         }
      },
      "platform": {
         "title": "Platform",
         "type": "string"
      },
      "inputs": {
         "title": "Inputs",
         "type": "array",
         "items": {
            "$ref": "#/definitions/MetadataTensor"
         }
      },
      "outputs": {
         "title": "Outputs",
         "type": "array",
         "items": {
            "$ref": "#/definitions/MetadataTensor"
         }
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      }
   },
   "required": [
```


(continues on next page)

(continued from previous page)


```
      "name",
      "platform"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "content_type": {
               "title": "Content Type",
               "type": "string"
            },
            "headers": {
               "title": "Headers",
               "type": "object"
            }
         }
      },
      "MetadataTensor": {
         "title": "MetadataTensor",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "datatype": {
               "title": "Datatype",
               "type": "string"
            },
            "shape": {
               "title": "Shape",
               "type": "array",
               "items": {
                  "type": "integer"
               }
            },
            "parameters": {
               "$ref": "#/definitions/Parameters"
            }
         },
         "required": [
            "name",
            "datatype",
            "shape"
         ]
```


(continues on next page)

(continued from previous page)

```
      }
   }
}
```

Fields

- inputs (List[mlserver.types.dataplane.MetadataTensor] | None)
- name (str)
- outputs (List[mlserver.types.dataplane.MetadataTensor] | None)
- parameters (mlserver.types.dataplane.Parameters | None)
- platform (str)
- versions (List[str] | None)

field inputs: List[*MetadataTensor*] | None = None

field name: str [Required]

field outputs: List[*MetadataTensor*] | None = None

field parameters: *Parameters* | None = None

field platform: str [Required]

field versions: List[str] | None = None

pydantic model mlserver.types.**MetadataServerErrorResponse**

```
{
   "title": "MetadataServerErrorResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "error": {
         "title": "Error",
         "type": "string"
      }
   },
   "required": [
      "error"
   ]
}
```

Fields

- error (str)

field error: str [Required]

pydantic model mlserver.types.**MetadataServerResponse**


```
{
   "title": "MetadataServerResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "version": {
         "title": "Version",
         "type": "string"
      },
      "extensions": {
         "title": "Extensions",
         "type": "array",
         "items": {
            "type": "string"
         }
      }
   },
   "required": [
      "name",
      "version",
      "extensions"
   ]
}
```


Fields

- extensions (List[str])
- name (str)
- version (str)

field extensions: List[str] [Required]

field name: str [Required]

field version: str [Required]

pydantic model mlserver.types.**MetadataTensor**


```
{
   "title": "MetadataTensor",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
```


(continues on next page)

(continued from previous page)


```
      "name": {
         "title": "Name",
         "type": "string"
      },
      "datatype": {
         "title": "Datatype",
         "type": "string"
      },
      "shape": {
         "title": "Shape",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      }
   },
   "required": [
      "name",
      "datatype",
      "shape"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "content_type": {
               "title": "Content Type",
               "type": "string"
            },
            "headers": {
               "title": "Headers",
               "type": "object"
            }
         }
      }
   }
}
```


Fields

- datatype (str)
- name (str)
- parameters (mlserver.types.dataplane.Parameters | None)
- shape (List[int])

field datatype: str [Required]

field name: str [Required]

field parameters: *Parameters* | None = None

field shape: List[int] [Required]

pydantic model mlserver.types.**Parameters**


```
{
   "title": "Parameters",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "content_type": {
         "title": "Content Type",
         "type": "string"
      },
      "headers": {
         "title": "Headers",
         "type": "object"
      }
   }
}
```


Config

- **extra**: *Extra = allow*

Fields

- content_type (str | None)
- headers (Dict[str, Any] | None)

field content_type: str | None = None

field headers: Dict[str, Any] | None = None

pydantic model mlserver.types.**RepositoryIndexRequest**


```
{
   "title": "RepositoryIndexRequest",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "ready": {
         "title": "Ready",
         "type": "boolean"
      }
   }
}
```

Fields

- ready (bool | None)

field ready: bool | None = None

pydantic model mlserver.types.**RepositoryIndexResponse**


```
{
   "title": "RepositoryIndexResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "array",
   "items": {
      "$ref": "#/definitions/RepositoryIndexResponseItem"
   },
   "definitions": {
      "State": {
         "title": "State",
         "description": "An enumeration.",
         "enum": [
            "UNKNOWN",
            "READY",
            "UNAVAILABLE",
            "LOADING",
            "UNLOADING"
         ]
      },
      "RepositoryIndexResponseItem": {
         "title": "RepositoryIndexResponseItem",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "name": {
               "title": "Name",
               "type": "string"
            },
            "version": {
               "title": "Version",
               "type": "string"
            },
            "state": {
               "$ref": "#/definitions/State"
            },
            "reason": {
               "title": "Reason",
               "type": "string"
            }
         },
         "required": [
            "name",
```


(continues on next page)

(continued from previous page)

```
            "state",
            "reason"
         ]
      }
   }
}
```

Fields

- `__root__ (List[mlserver.types.model_repository.RepositoryIndexResponseItem])`

pydantic model mlserver.types.**RepositoryIndexResponseItem**

```
{
   "title": "RepositoryIndexResponseItem",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "version": {
         "title": "Version",
         "type": "string"
      },
      "state": {
         "$ref": "#/definitions/State"
      },
      "reason": {
         "title": "Reason",
         "type": "string"
      }
   },
   "required": [
      "name",
      "state",
      "reason"
   ],
   "definitions": {
      "State": {
         "title": "State",
         "description": "An enumeration.",
         "enum": [
            "UNKNOWN",
            "READY",
            "UNAVAILABLE",
            "LOADING",
            "UNLOADING"
```

(continues on next page)

(continued from previous page)

```
            ]
        }
    }
}
```

Fields

- name (str)
- reason (str)
- state (mlserver.types.model_repository.State)
- version (str | None)

field name: str [Required]

field reason: str [Required]

field state: *State* [Required]

field version: str | None = None

pydantic model mlserver.types.**RepositoryLoadErrorResponse**

```
{
   "title": "RepositoryLoadErrorResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "error": {
         "title": "Error",
         "type": "string"
      }
   }
}
```

Fields

- error (str | None)

field error: str | None = None

pydantic model mlserver.types.**RepositoryUnloadErrorResponse**

```
{
   "title": "RepositoryUnloadErrorResponse",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "error": {
```

(continues on next page)

(continued from previous page)

```
            "title": "Error",
            "type": "string"
        }
    }
}
```

Fields

- error (str | None)

field error: str | None = None

pydantic model mlserver.types.**RequestInput**

```
{
   "title": "RequestInput",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
         "type": "string"
      },
      "shape": {
         "title": "Shape",
         "type": "array",
         "items": {
            "type": "integer"
         }
      },
      "datatype": {
         "title": "Datatype",
         "type": "string"
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      },
      "data": {
         "$ref": "#/definitions/TensorData"
      }
   },
   "required": [
      "name",
      "shape",
      "datatype",
      "data"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
```

(continues on next page)

(continued from previous page)


```
        "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
        "type": "object",
        "properties": {
           "content_type": {
              "title": "Content Type",
              "type": "string"
           },
           "headers": {
              "title": "Headers",
              "type": "object"
           }
        }
     },
     "TensorData": {
        "title": "TensorData",
        "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525"
     }
  }
}
```


Fields

- data (mlserver.types.dataplane.TensorData)
- datatype (str)
- name (str)
- parameters (mlserver.types.dataplane.Parameters | None)
- shape (List[int])

field data: *TensorData* **[Required]**

field datatype: **str [Required]**

field name: **str [Required]**

field parameters: *Parameters* **| None = None**

field shape: **List[int] [Required]**

pydantic model mlserver.types.**RequestOutput**


```
{
   "title": "RequestOutput",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
```


(continues on next page)

(continued from previous page)


```
      "name": {
         "title": "Name",
         "type": "string"
      },
      "parameters": {
         "$ref": "#/definitions/Parameters"
      }
   },
   "required": [
      "name"
   ],
   "definitions": {
      "Parameters": {
         "title": "Parameters",
         "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
         "type": "object",
         "properties": {
            "content_type": {
               "title": "Content Type",
               "type": "string"
            },
            "headers": {
               "title": "Headers",
               "type": "object"
            }
         }
      }
   }
}
```


Fields

- name (str)
- parameters (mlserver.types.dataplane.Parameters | None)

field name: str [Required]

field parameters: *Parameters* | None = None

pydantic model mlserver.types.**ResponseOutput**


```
{
   "title": "ResponseOutput",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
   "type": "object",
   "properties": {
      "name": {
         "title": "Name",
```


(continues on next page)

(continued from previous page)


```
            "type": "string"
         },
         "shape": {
            "title": "Shape",
            "type": "array",
            "items": {
               "type": "integer"
            }
         },
         "datatype": {
            "title": "Datatype",
            "type": "string"
         },
         "parameters": {
            "$ref": "#/definitions/Parameters"
         },
         "data": {
            "$ref": "#/definitions/TensorData"
         }
      },
      "required": [
         "name",
         "shape",
         "datatype",
         "data"
      ],
      "definitions": {
         "Parameters": {
            "title": "Parameters",
            "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525",
            "type": "object",
            "properties": {
               "content_type": {
                  "title": "Content Type",
                  "type": "string"
               },
               "headers": {
                  "title": "Headers",
                  "type": "object"
               }
            }
         },
         "TensorData": {
            "title": "TensorData",
            "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525"
         }
      }
}
```

Fields

- data (mlserver.types.dataplane.TensorData)
- datatype (str)
- name (str)
- parameters (mlserver.types.dataplane.Parameters | None)
- shape (List[int])

field data: *TensorData* **[Required]**

field datatype: **str [Required]**

field name: **str [Required]**

field parameters: *Parameters* **| None = None**

field shape: **List[int] [Required]**

class mlserver.types.**State**(*value*)

An enumeration.

pydantic model mlserver.types.**TensorData**

```
{
   "title": "TensorData",
   "description": "Override Pydantic's BaseModel class to ensure all payloads␣
↪exclude unset\nfields by default.\n\nFrom:\n    https://github.com/pydantic/
↪pydantic/issues/1387#issuecomment-612901525"
}
```

Fields

- __root__ (Any)

9.4.3 Codecs

Codecs are used to encapsulate the logic required to encode / decode payloads following the Open Inference Protocol into high-level Python types. You can read more about the high-level concepts behind codecs in the *Content Types (and Codecs)* section of the docs, as well as how to use them.

Base Codecs

All the codecs within MLServer extend from either the *InputCodec* or the *RequestCodec* base classes. These define the interface to deal with input (outputs) and request (responses) respectively.

class mlserver.codecs.**InputCodec**

The InputCodec interface lets you define type conversions of your raw input data to / from the Open Inference Protocol. Note that this codec applies at the individual input (output) level.

For request-wide transformations (e.g. dataframes), use the RequestCodec interface instead.

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_input(*request_input:* RequestInput) → Any

Decode a request input into a high-level Python type.

classmethod decode_output(*response_output:* ResponseOutput) → Any

Decode a response output into a high-level Python type.

classmethod encode_input(*name: str*, *payload: Any*, ***kwargs*) → *RequestInput*

Encode the given payload into a `RequestInput`.

classmethod encode_output(*name: str*, *payload: Any*, ***kwargs*) → *ResponseOutput*

Encode the given payload into a response output.

class mlserver.codecs.**RequestCodec**

The `RequestCodec` interface lets you define request-level conversions between high-level Python types and the Open Inference Protocol. This can be useful where the encoding of your payload encompases multiple input heads (e.g. dataframes, where each column can be thought as a separate input head).

For individual input-level encoding / decoding, use the `InputCodec` interface instead.

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_request(*request:* InferenceRequest) → Any

Decode an inference request into a high-level Python object.

classmethod decode_response(*response:* InferenceResponse) → Any

Decode an inference response into a high-level Python object.

classmethod encode_request(*payload: Any*, ***kwargs*) → *InferenceRequest*

Encode the given payload into an inference request.

classmethod encode_response(*model_name: str*, *payload: Any*, *model_version: str | None = None*, ***kwargs*) → *InferenceResponse*

Encode the given payload into an inference response.

Built-in Codecs

The `mlserver` package will include a set of built-in codecs to cover common conversions. You can learn more about these in the *Available Content Types* section of the docs.

class mlserver.codecs.**Base64Codec**

Codec that convers to / from a base64 input.

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_input(*request_input:* RequestInput) → List[bytes]

Decode a request input into a high-level Python type.

classmethod decode_output(*response_output:* ResponseOutput) → List[bytes]

Decode a response output into a high-level Python type.

classmethod encode_input(*name: str*, *payload: List[bytes]*, *use_bytes: bool = True*, ***kwargs*) → *RequestInput*

Encode the given payload into a `RequestInput`.

classmethod encode_output(*name: str*, *payload: List[bytes]*, *use_bytes: bool = True*, ***kwargs*) → *ResponseOutput*

Encode the given payload into a response output.

class mlserver.codecs.**DatetimeCodec**

Codec that convers to / from a datetime input.

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_input(*request_input:* RequestInput) → List[datetime]

Decode a request input into a high-level Python type.

classmethod decode_output(*response_output:* ResponseOutput) → List[datetime]

Decode a response output into a high-level Python type.

classmethod encode_input(*name: str*, *payload: List[str | datetime]*, *use_bytes: bool = True*, ***kwargs*) → *RequestInput*

Encode the given payload into a `RequestInput`.

classmethod encode_output(*name: str*, *payload: List[str | datetime]*, *use_bytes: bool = True*, ***kwargs*) → *ResponseOutput*

Encode the given payload into a response output.

class mlserver.codecs.**NumpyCodec**

Decodes an request input (response output) as a NumPy array.

TypeHint

alias of `ndarray`

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_input(*request_input:* RequestInput) → ndarray

Decode a request input into a high-level Python type.

classmethod decode_output(*response_output:* ResponseOutput) → ndarray

Decode a response output into a high-level Python type.

classmethod encode_input(*name: str*, *payload: ndarray*, ***kwargs*) → *RequestInput*

Encode the given payload into a `RequestInput`.

classmethod encode_output(*name: str*, *payload: ndarray*, ***kwargs*) → *ResponseOutput*

Encode the given payload into a response output.

class mlserver.codecs.**NumpyRequestCodec**

Decodes the first input (output) of request (response) as a NumPy array. This codec can be useful for cases where the whole payload is a single NumPy tensor.

InputCodec

alias of *NumpyCodec*

class mlserver.codecs.**PandasCodec**

Decodes a request (response) into a Pandas DataFrame, assuming each input (output) head corresponds to a column of the DataFrame.

TypeHint

alias of `DataFrame`

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_request(*request:* InferenceRequest) → DataFrame

Decode an inference request into a high-level Python object.

classmethod decode_response(*response:* InferenceResponse) → DataFrame

Decode an inference response into a high-level Python object.

classmethod encode_request(*payload: DataFrame*, *use_bytes: bool = True*, ***kwargs*) → *InferenceRequest*

Encode the given payload into an inference request.

classmethod encode_response(*model_name: str*, *payload: DataFrame*, *model_version: str | None = None*, *use_bytes: bool = True*, ***kwargs*) → *InferenceResponse*

Encode the given payload into an inference response.

class mlserver.codecs.**StringCodec**

Encodes a list of Python strings as a BYTES input (output).

classmethod can_encode(*payload: Any*) → bool

Evaluate whether the codec can encode (decode) the payload.

classmethod decode_input(*request_input:* RequestInput) → List[str]

Decode a request input into a high-level Python type.

classmethod decode_output(*response_output:* ResponseOutput) → List[str]

Decode a response output into a high-level Python type.

classmethod encode_input(*name: str*, *payload: List[str]*, *use_bytes: bool = True*, ***kwargs*) → *RequestInput*

Encode the given payload into a `RequestInput`.

classmethod encode_output(*name: str*, *payload: List[str]*, *use_bytes: bool = True*, ***kwargs*) → *ResponseOutput*

Encode the given payload into a response output.

class mlserver.codecs.**StringRequestCodec**

Decodes the first input (output) of request (response) as a list of strings. This codec can be useful for cases where the whole payload is a single list of strings.

InputCodec

alias of *StringCodec*

9.4.4 Metrics

The MLServer package exposes a set of methods that let you register and track custom metrics. This can be used within your own *custom inference runtimes*. To learn more about how to expose custom metrics, check out the *metrics usage guide*.

mlserver.**log**(***metrics*)

Logs a new set of metric values. Each kwarg of this method will be treated as a separate metric / value pair. If any of the metrics does not exist, a new one will be created with a default description.

mlserver.**register**(*name: str*, *description: str*) → Histogram

Registers a new metric with its description. If the metric already exists, it will just return the existing one.

CHAPTER

TEN

EXAMPLES

To see MLServer in action you can check out the examples below. These are end-to-end notebooks, showing how to serve models with MLServer.

10.1 Inference Runtimes

If you are interested in how MLServer interacts with particular model frameworks, you can check the following examples. These focus on showcasing the different *inference runtimes* that ship with MLServer out of the box. Note that, for **advanced use cases**, you can also write your own custom inference runtime (see the *example below on custom models*).

- *Serving Scikit-Learn models*
- *Serving XGBoost models*
- *Serving LightGBM models*
- *Serving Tempo pipelines*
- *Serving MLflow models*
- *Serving custom models*
- *Serving Alibi Detect models*
- *Serving HuggingFace models*

10.1.1 Serving Scikit-Learn models

Out of the box, `mlserver` supports the deployment and serving of `scikit-learn` models. By default, it will assume that these models have been serialised using `joblib`.

In this example, we will cover how we can train and serialise a simple model, to then serve it using `mlserver`.

Training

The first step will be to train a simple `scikit-learn` model. For that, we will use the MNIST example from the `scikit-learn` documentation which trains an SVM model.

```
# Original source code and more details can be found in:
# https://scikit-learn.org/stable/auto_examples/classification/plot_digits_
↪classification.html

# Import datasets, classifiers and performance metrics
```

(continues on next page)

(continued from previous page)

```
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)
```

Saving our trained model

To save our trained model, we will serialise it using `joblib`. While this is not a perfect approach, it's currently the recommended method to persist models to disk in the `scikit-learn` documentation.

Our model will be persisted as a file named `mnist-svm.joblib`

```
import joblib

model_file_name = "mnist-svm.joblib"
joblib.dump(classifier, model_file_name)
```

Serving

Now that we have trained and saved our model, the next step will be to serve it using `mlserver`. For that, we will need to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

settings.json

```
%%writefile settings.json
{
    "debug": "true"
}
```

model-settings.json

```
%%writefile model-settings.json
{
    "name": "mnist-svm",
    "implementation": "mlserver_sklearn.SKLearnModel",
    "parameters": {
        "uri": "./mnist-svm.joblib",
        "version": "v0.1.0"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = X_test[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
endpoint = "http://localhost:8080/v2/models/mnist-svm/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

As we can see above, the model predicted the input as the number 8, which matches what's on the test set.

```
y_test[0]
```

```

```

10.1.2 Serving XGBoost models

Out of the box, `mlserver` supports the deployment and serving of `xgboost` models. By default, it will assume that these models have been serialised using the `bst.save_model()` method.

In this example, we will cover how we can train and serialise a simple model, to then serve it using `mlserver`.

Training

The first step will be to train a simple `xgboost` model. For that, we will use the mushrooms example from the `xgboost` Getting Started guide.

```
# Original code and extra details can be found in:
# https://xgboost.readthedocs.io/en/latest/get_started.html#python

import os
import xgboost as xgb
import requests

from urllib.parse import urlparse
from sklearn.datasets import load_svmlight_file


TRAIN_DATASET_URL = 'https://raw.githubusercontent.com/dmlc/xgboost/master/demo/data/
↪agaricus.txt.train'
TEST_DATASET_URL = 'https://raw.githubusercontent.com/dmlc/xgboost/master/demo/data/
↪agaricus.txt.test'


def _download_file(url: str) -> str:
    parsed = urlparse(url)
    file_name = os.path.basename(parsed.path)
    file_path = os.path.join(os.getcwd(), file_name)

    res = requests.get(url)

    with open(file_path, 'wb') as file:
        file.write(res.content)
```

(continues on next page)

(continued from previous page)

```
    return file_path

train_dataset_path = _download_file(TRAIN_DATASET_URL)
test_dataset_path = _download_file(TEST_DATASET_URL)

# NOTE: Workaround to load SVMLight files from the XGBoost example
X_train, y_train = load_svmlight_file(train_dataset_path)
X_test, y_test = load_svmlight_file(test_dataset_path)
X_train = X_train.toarray()
X_test = X_test.toarray()

# read in data
dtrain = xgb.DMatrix(data=X_train, label=y_train)

# specify parameters via map
param = {'max_depth':2, 'eta':1, 'objective':'binary:logistic' }
num_round = 2
bst = xgb.train(param, dtrain, num_round)

bst
```

Saving our trained model

To save our trained model, we will serialise it using `bst.save_model()` and the JSON format. This is the approach by the XGBoost project.

Our model will be persisted as a file named `mushroom-xgboost.json`.

```
model_file_name = 'mushroom-xgboost.json'
bst.save_model(model_file_name)
```

Serving

Now that we have trained and saved our model, the next step will be to serve it using `mlserver`. For that, we will need to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

`settings.json`

```
%%writefile settings.json
{
    "debug": "true"
}
```

model-settings.json

```
%%writefile model-settings.json
{
    "name": "mushroom-xgboost",
    "implementation": "mlserver_xgboost.XGBoostModel",
    "parameters": {
        "uri": "./mushroom-xgboost.json",
        "version": "v0.1.0"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = X_test[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/mushroom-xgboost/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

As we can see above, the model predicted the input as close to `0`, which matches what's on the test set.

```
y_test[0]
```

10.1.3 Serving LightGBM models

Out of the box, `mlserver` supports the deployment and serving of `lightgbm` models. By default, it will assume that these models have been serialised using the `bst.save_model()` method.

In this example, we will cover how we can train and serialise a simple model, to then serve it using `mlserver`.

Training

To test the LightGBM Server, first we need to generate a simple LightGBM model using Python.

```
import lightgbm as lgb
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import os

model_dir = "."
BST_FILE = "iris-lightgbm.bst"

iris = load_iris()
y = iris['target']
X = iris['data']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
dtrain = lgb.Dataset(X_train, label=y_train)

params = {
    'objective':'multiclass',
    'metric':'softmax',
    'num_class': 3
}
lgb_model = lgb.train(params=params, train_set=dtrain)
model_file = os.path.join(model_dir, BST_FILE)
lgb_model.save_model(model_file)
```

Our model will be persisted as a file named `iris-lightgbm.bst`.

Serving

Now that we have trained and saved our model, the next step will be to serve it using `mlserver`. For that, we will need to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

settings.json

```
%%writefile settings.json
{
    "debug": "true"
}
```

model-settings.json

```
%%writefile model-settings.json
{
    "name": "iris-lgb",
    "implementation": "mlserver_lightgbm.LightGBMModel",
    "parameters": {
        "uri": "./iris-lightgbm.bst",
        "version": "v0.1.0"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = X_test[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict-prob",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
endpoint = "http://localhost:8080/v2/models/iris-lgb/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

As we can see above, the model predicted the probability for each class, and the probability of class 1 is the biggest, close to `0.99`, which matches what's on the test set.

```
y_test[0]
```

10.1.4 Running a Tempo pipeline in MLServer

This example walks you through how to create and serialise a Tempo pipeline, which can then be served through MLServer. This pipeline can contain custom Python arbitrary code.

Creating the pipeline

The first step will be to create our Tempo pipeline.

```
import numpy as np
import os

from tempo import ModelFramework, Model, Pipeline, pipeline
from tempo.seldon import SeldonDockerRuntime
from tempo.kfserving import KFServingV2Protocol


MODELS_PATH = os.path.join(os.getcwd(), 'models')

docker_runtime = SeldonDockerRuntime()

sklearn_iris_path = os.path.join(MODELS_PATH, 'sklearn-iris')
sklearn_model = Model(
    name="test-iris-sklearn",
    runtime=docker_runtime,
    platform=ModelFramework.SKLearn,
    uri="gs://seldon-models/sklearn/iris",
    local_folder=sklearn_iris_path,
)

xgboost_iris_path = os.path.join(MODELS_PATH, 'xgboost-iris')
xgboost_model = Model(
    name="test-iris-xgboost",
    runtime=docker_runtime,
    platform=ModelFramework.XGBoost,
    uri="gs://seldon-models/xgboost/iris",
    local_folder=xgboost_iris_path,
)
```

(continues on next page)

(continued from previous page)

```
inference_pipeline_path = os.path.join(MODELS_PATH, 'inference-pipeline')
@pipeline(
    name="inference-pipeline",
    models=[sklearn_model, xgboost_model],
    runtime=SeldonDockerRuntime(protocol=KFServingV2Protocol()),
    local_folder=inference_pipeline_path
)
def inference_pipeline(payload: np.ndarray) -> np.ndarray:
    res1 = sklearn_model(payload)
    if res1[0][0] > 0.7:
        return res1
    else:
        return xgboost_model(payload)
```

This pipeline can then be serialised using `cloudpickle`.

```
inference_pipeline.save(save_env=False)
```

Serving the pipeline

Once we have our pipeline created and serialised, we can then create a `model-settings.json` file. This configuration file will hold the configuration specific to our MLOps pipeline.

```
%%writefile ./model-settings.json
{
    "name": "inference-pipeline",
    "implementation": "tempo.mlserver.InferenceRuntime",
    "parameters": {
        "uri": "./models/inference-pipeline"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Deploy our pipeline components

Additionally, we will also need to deploy our pipeline components. That is, the SKLearn and XGBoost models. We can do that as:

```
inference_pipeline.deploy()
```

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = np.array([[0.1, 3.1, 1.5, 0.2]])
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/inference-pipeline/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

```

```

10.1.5 Serving MLflow models

Out of the box, MLServer supports the deployment and serving of MLflow models with the following features:

- Loading of MLflow Model artifacts.
- Support of dataframes, dict-of-tensors and tensor inputs.

In this example, we will showcase some of this features using an example model.

```
from IPython.core.magic import register_line_cell_magic

@register_line_cell_magic
def writetemplate(line, cell):
    with open(line, 'w') as f:
        f.write(cell.format(**globals()))
```

Training

The first step will be to train and serialise a MLflow model. For that, we will use the linear regression examle from the MLflow docs.

```
# %load src/train.py
# Original source code and more details can be found in:
# https://www.mlflow.org/docs/latest/tutorials-and-examples/tutorial.html

# The data set used in this example is from
# http://archive.ics.uci.edu/ml/datasets/Wine+Quality
# P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
# Modeling wine preferences by data mining from physicochemical properties.
# In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

import warnings
import sys

import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ElasticNet
from urllib.parse import urlparse
import mlflow
import mlflow.sklearn
from mlflow.models.signature import infer_signature

import logging

logging.basicConfig(level=logging.WARN)
logger = logging.getLogger(__name__)


def eval_metrics(actual, pred):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mae = mean_absolute_error(actual, pred)
    r2 = r2_score(actual, pred)
    return rmse, mae, r2


if __name__ == "__main__":
    warnings.filterwarnings("ignore")
    np.random.seed(40)

    # Read the wine-quality csv file from the URL
    csv_url = (
        "http://archive.ics.uci.edu/ml"
        "/machine-learning-databases/wine-quality/winequality-red.csv"
    )
    try:
        data = pd.read_csv(csv_url, sep=";")
    except Exception as e:
```

(continues on next page)

(continued from previous page)

```
        logger.exception(
            "Unable to download training & test CSV, "
            "check your internet connection. Error: %s",
            e,
        )

    # Split the data into training and test sets. (0.75, 0.25) split.
    train, test = train_test_split(data)

    # The predicted column is "quality" which is a scalar from [3, 9]
    train_x = train.drop(["quality"], axis=1)
    test_x = test.drop(["quality"], axis=1)
    train_y = train[["quality"]]
    test_y = test[["quality"]]

    alpha = float(sys.argv[1]) if len(sys.argv) > 1 else 0.5
    l1_ratio = float(sys.argv[2]) if len(sys.argv) > 2 else 0.5

    with mlflow.start_run():
        lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
        lr.fit(train_x, train_y)

        predicted_qualities = lr.predict(test_x)

        (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)

        print("Elasticnet model (alpha=%f, l1_ratio=%f):" % (alpha, l1_ratio))
        print("  RMSE: %s" % rmse)
        print("  MAE: %s" % mae)
        print("  R2: %s" % r2)

        mlflow.log_param("alpha", alpha)
        mlflow.log_param("l1_ratio", l1_ratio)
        mlflow.log_metric("rmse", rmse)
        mlflow.log_metric("r2", r2)
        mlflow.log_metric("mae", mae)

        tracking_url_type_store = urlparse(mlflow.get_tracking_uri()).scheme
        model_signature = infer_signature(train_x, train_y)

        # Model registry does not work with file store
        if tracking_url_type_store != "file":

            # Register the model
            # There are other ways to use the Model Registry,
            # which depends on the use case,
            # please refer to the doc for more information:
            # https://mlflow.org/docs/latest/model-registry.html#api-workflow
            mlflow.sklearn.log_model(
                lr,
                "model",
                registered_model_name="ElasticnetWineModel",
```

(continues on next page)

(continued from previous page)

```
                signature=model_signature,
            )
        else:
            mlflow.sklearn.log_model(lr, "model", signature=model_signature)
```

```
!python src/train.py
```

The training script will also serialise our trained model, leveraging the MLflow Model format. By default, we should be able to find the saved artifact under the `mlruns` folder.

```
import os

[experiment_file_path] = !ls -td ./mlruns/0/* | head -1
model_path = os.path.join(experiment_file_path, "artifacts", "model")
print(model_path)
```

```
!ls {model_path}
```

Serving

Now that we have trained and serialised our model, we are ready to start serving it. For that, the initial step will be to set up a `model-settings.json` that instructs MLServer to load our artifact using the MLflow Inference Runtime.

```
%%writetemplate ./model-settings.json
{{
    "name": "wine-classifier",
    "implementation": "mlserver_mlflow.MLflowRuntime",
    "parameters": {{
        "uri": "{model_path}"
    }}
}}
```

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set. For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

Note that, the request specifies the value `pd` as its *content type*, whereas every input specifies the *content type* `np`. These parameters will instruct MLServer to:

- Convert every input value to a NumPy array, using the data type and shape information provided.

- Group all the different inputs into a Pandas DataFrame, using their names as the column names.

To learn more about how MLServer uses content type parameters, you can check this *worked out example*.

```
import requests

inference_request = {
    "inputs": [
        {
          "name": "fixed acidity",
          "shape": [1],
          "datatype": "FP32",
          "data": [7.4],
        },
        {
          "name": "volatile acidity",
          "shape": [1],
          "datatype": "FP32",
          "data": [0.7000],
        },
        {
          "name": "citric acid",
          "shape": [1],
          "datatype": "FP32",
          "data": [0],
        },
        {
          "name": "residual sugar",
          "shape": [1],
          "datatype": "FP32",
          "data": [1.9],
        },
        {
          "name": "chlorides",
          "shape": [1],
          "datatype": "FP32",
          "data": [0.076],
        },
        {
          "name": "free sulfur dioxide",
          "shape": [1],
          "datatype": "FP32",
          "data": [11],
        },
        {
          "name": "total sulfur dioxide",
          "shape": [1],
          "datatype": "FP32",
          "data": [34],
        },
        {
          "name": "density",
          "shape": [1],
          "datatype": "FP32",
```

(continues on next page)

(continued from previous page)

```
          "data": [0.9978],
        },
        {
          "name": "pH",
          "shape": [1],
          "datatype": "FP32",
          "data": [3.51],
        },
        {
          "name": "sulphates",
          "shape": [1],
          "datatype": "FP32",
          "data": [0.56],
        },
        {
          "name": "alcohol",
          "shape": [1],
          "datatype": "FP32",
          "data": [9.4],
        },
    ]
}

endpoint = "http://localhost:8080/v2/models/wine-classifier/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

As we can see in the output above, the predicted quality score for our input wine was `5.57`.

MLflow Scoring Protocol

MLflow currently ships with an scoring server with its own protocol. In order to provide a drop-in replacement, the MLflow runtime in MLServer also exposes a custom endpoint which matches the signature of the MLflow's `/invocations` endpoint.

As an example, we can try to send the same request that sent previously, but using MLflow's protocol. Note that, in both cases, the request will be handled by the same MLServer instance.

```
import requests

inference_request = {
    "dataframe_split": {
        "columns": [
            "alcohol",
            "chlorides",
            "citric acid",
            "density",
            "fixed acidity",
            "free sulfur dioxide",
            "pH",
            "residual sugar",
```

(continues on next page)

(continued from previous page)

```
            "sulphates",
            "total sulfur dioxide",
            "volatile acidity",
        ],
        "data": [[7.4,0.7,0,1.9,0.076,11,34,0.9978,3.51,0.56,9.4]]
    }
}

endpoint = "http://localhost:8080/invocations"
response = requests.post(endpoint, json=inference_request)

response.json()
```

As we can see above, the predicted quality for our input is `5.57`, matching the prediction we obtained above.

MLflow Model Signature

MLflow lets users define a *model signature*, where they can specify what types of inputs does the model accept, and what types of outputs it returns. Similarly, the V2 inference protocol employed by MLServer defines a *metadata endpoint* which can be used to query what inputs and outputs does the model accept. However, even though they serve similar functions, the data schemas used by each one of them are not compatible between them.

To solve this, if your model defines a MLflow model signature, MLServer will convert *on-the-fly* this signature to a metadata schema compatible with the V2 Inference Protocol. This will also include specifying any extra *content type* that is required to correctly decode / encode your data.

As an example, we can first have a look at the model signature saved for our MLflow model. This can be seen directly on the `MLModel` file saved by our model.

```
!cat {model_path}/MLmodel
```

We can then query the metadata endpoint, to see the model metadata inferred by MLServer from our test model's signature. For this, we will use the `/v2/models/wine-classifier/` endpoint.

```
import requests


endpoint = "http://localhost:8080/v2/models/wine-classifier"
response = requests.get(endpoint)

response.json()
```

As we should be able to see, the model metadata now matches the information contained in our model signature, including any extra content types necessary to decode our data correctly.

```

```

10.1.6 Serving a custom model

The `mlserver` package comes with inference runtime implementations for `scikit-learn` and `xgboost` models. However, some times we may also need to roll out our own inference server, with custom logic to perform inference. To support this scenario, MLServer makes it really easy to create your own extensions, which can then be containerised and deployed in a production environment.

Overview

In this example, we will train a numpyro model. The `numpyro` library streamlines the implementation of probabilistic models, abstracting away advanced inference and training algorithms.

Out of the box, `mlserver` doesn't provide an inference runtime for `numpyro`. However, through this example we will see how easy is to develop our own.

Training

The first step will be to train our model. This will be a very simple bayesian regression model, based on an example provided in the numpyro docs.

Since this is a probabilistic model, during training we will compute an approximation to the posterior distribution of our model using MCMC.

```
# Original source code and more details can be found in:
# https://nbviewer.jupyter.org/github/pyro-ppl/numpyro/blob/master/notebooks/source/
↪bayesian_regression.ipynb


import numpyro
import numpy as np
import pandas as pd

from numpyro import distributions as dist
from jax import random
from numpyro.infer import MCMC, NUTS

DATASET_URL = "https://raw.githubusercontent.com/rmcelreath/rethinking/master/data/
↪WaffleDivorce.csv"
dset = pd.read_csv(DATASET_URL, sep=";")

standardize = lambda x: (x - x.mean()) / x.std()

dset["AgeScaled"] = dset.MedianAgeMarriage.pipe(standardize)
dset["MarriageScaled"] = dset.Marriage.pipe(standardize)
dset["DivorceScaled"] = dset.Divorce.pipe(standardize)


def model(marriage=None, age=None, divorce=None):
    a = numpyro.sample("a", dist.Normal(0.0, 0.2))
    M, A = 0.0, 0.0
    if marriage is not None:
        bM = numpyro.sample("bM", dist.Normal(0.0, 0.5))
        M = bM * marriage
```

(continues on next page)

(continued from previous page)

```
    if age is not None:
        bA = numpyro.sample("bA", dist.Normal(0.0, 0.5))
        A = bA * age
    sigma = numpyro.sample("sigma", dist.Exponential(1.0))
    mu = a + M + A
    numpyro.sample("obs", dist.Normal(mu, sigma), obs=divorce)


# Start from this source of randomness. We will split keys for subsequent operations.
rng_key = random.PRNGKey(0)
rng_key, rng_key_ = random.split(rng_key)

num_warmup, num_samples = 1000, 2000

# Run NUTS.
kernel = NUTS(model)
mcmc = MCMC(kernel, num_warmup=num_warmup, num_samples=num_samples)
mcmc.run(
    rng_key_, marriage=dset.MarriageScaled.values, divorce=dset.DivorceScaled.values
)
mcmc.print_summary()
```

Saving our trained model

Now that we have *trained* our model, the next step will be to save it so that it can be loaded afterwards at serving-time. Note that, since this is a probabilistic model, we will only need to save the traces that approximate the posterior distribution over latent parameters.

This will get saved in a `numpyro-divorce.json` file.

```
import json

samples = mcmc.get_samples()
serialisable = {}
for k, v in samples.items():
    serialisable[k] = np.asarray(v).tolist()

model_file_name = "numpyro-divorce.json"
with open(model_file_name, "w") as model_file:
    json.dump(serialisable, model_file)
```

Serving

The next step will be to serve our model using `mlserver`. For that, we will first implement an extension which serve as the *runtime* to perform inference using our custom `numpyro` model.

Custom inference runtime

Our custom inference wrapper should be responsible of:

- Loading the model from the set samples we saved previously.
- Running inference using our model structure, and the posterior approximated from the samples.

```
# %load models.py
import json
import numpyro
import numpy as np

from jax import random
from mlserver import MLModel
from mlserver.codecs import decode_args
from mlserver.utils import get_model_uri
from numpyro.infer import Predictive
from numpyro import distributions as dist
from typing import Optional


class NumpyroModel(MLModel):
    async def load(self) -> bool:
        model_uri = await get_model_uri(self._settings)
        with open(model_uri) as model_file:
            raw_samples = json.load(model_file)

        self._samples = {}
        for k, v in raw_samples.items():
            self._samples[k] = np.array(v)

        self._predictive = Predictive(self._model, self._samples)

        return True

    @decode_args
    async def predict(
        self,
        marriage: Optional[np.ndarray] = None,
        age: Optional[np.ndarray] = None,
        divorce: Optional[np.ndarray] = None,
    ) -> np.ndarray:
        predictions = self._predictive(
            rng_key=random.PRNGKey(0), marriage=marriage, age=age, divorce=divorce
        )

        obs = predictions["obs"]
        obs_mean = obs.mean()

        return np.asarray(obs_mean)

    def _model(self, marriage=None, age=None, divorce=None):
        a = numpyro.sample("a", dist.Normal(0.0, 0.2))
```

(continues on next page)

(continued from previous page)

```
        M, A = 0.0, 0.0
        if marriage is not None:
            bM = numpyro.sample("bM", dist.Normal(0.0, 0.5))
            M = bM * marriage
        if age is not None:
            bA = numpyro.sample("bA", dist.Normal(0.0, 0.5))
            A = bA * age
        sigma = numpyro.sample("sigma", dist.Exponential(1.0))
        mu = a + M + A
        numpyro.sample("obs", dist.Normal(mu, sigma), obs=divorce)
```

Settings files

The next step will be to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

settings.json

```
# %load settings.json
{
    "debug": "true"
}
```

model-settings.json

```
# %load model-settings.json
{
    "name": "numpyro-divorce",
    "implementation": "models.NumpyroModel",
    "parameters": {
        "uri": "./numpyro-divorce.json"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests
import numpy as np

from mlserver.types import InferenceRequest
from mlserver.codecs import NumpyCodec

x_0 = np.array([28.0])
inference_request = InferenceRequest(
    inputs=[
        NumpyCodec.encode_input(name="marriage", payload=x_0)
    ]
)

endpoint = "http://localhost:8080/v2/models/numpyro-divorce/infer"
response = requests.post(endpoint, json=inference_request.dict())

response.json()
```

Deployment

Now that we have written and tested our custom model, the next step is to deploy it. With that goal in mind, the rough outline of steps will be to first build a custom image containing our code, and then deploy it.

Specifying requirements

MLServer will automatically find your requirements.txt file and install necessary python packages

```
# %load requirements.txt
numpy==1.22.4
numpyro==0.8.0
jax==0.2.24
jaxlib==0.3.7
```

Building a custom image

Note: This section expects that Docker is available and running in the background.

MLServer offers helpers to build a custom Docker image containing your code. In this example, we will use the `mlserver build` subcommand to create an image, which we'll be able to deploy later.

Note that this section expects that Docker is available and running in the background, as well as a functional cluster with Seldon Core installed and some familiarity with `kubectl`.

```
%%bash
mlserver build . -t 'my-custom-numpyro-server:0.1.0'
```

To ensure that the image is fully functional, we can spin up a container and then send a test request. To start the container, you can run something along the following lines in a separate terminal:

```
docker run -it --rm -p 8080:8080 my-custom-numpyro-server:0.1.0
```

```
import numpy as np

from mlserver.types import InferenceRequest
from mlserver.codecs import NumpyCodec

x_0 = np.array([28.0])
inference_request = InferenceRequest(
    inputs=[
        NumpyCodec.encode_input(name="marriage", payload=x_0)
    ]
)

endpoint = "http://localhost:8080/v2/models/numpyro-divorce/infer"
response = requests.post(endpoint, json=inference_request.dict())

response.json()
```

As we should be able to see, the server running within our Docker image responds as expected.

Deploying our custom image

Note: This section expects access to a functional Kubernetes cluster with Seldon Core installed and some familiarity with `kubectl`.

Now that we've built a custom image and verified that it works as expected, we can move to the next step and deploy it. There is a large number of tools out there to deploy images. However, for our example, we will focus on deploying it to a cluster running Seldon Core.

Note: Also consider that depending on your Kubernetes installation Seldon Core might expect to get the container image from a public container registry like Docker hub or Google Container Registry. For that you need to do an extra step of pushing the container to the registry using `docker tag <image name> <container registry>/<image`

name> and docker push <container registry>/<image name> and also updating the image section of the yaml file to <container registry>/<image name>.

For that, we will need to create a SeldonDeployment resource which instructs Seldon Core to deploy a model embedded within our custom image and compliant with the V2 Inference Protocol. This can be achieved by *applying* (i.e. kubectl apply) a SeldonDeployment manifest to the cluster, similar to the one below:

```
%%writefile seldondeployment.yaml
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: numpyro-model
spec:
  protocol: v2
  predictors:
    - name: default
      graph:
        name: numpyro-divorce
        type: MODEL
      componentSpecs:
        - spec:
            containers:
              - name: numpyro-divorce
                image: my-custom-numpyro-server:0.1.0
```

```

```

10.1.7 Serving Alibi-Detect models

Out of the box, mlserver supports the deployment and serving of alibi_detect models. Alibi Detect is an open source Python library focused on outlier, adversarial and drift detection. In this example, we will cover how we can create a detector configuration to then serve it using mlserver.

Fetch reference data

The first step will be to fetch a reference data and other relevant metadata for an alibi-detect model.

For that, we will use the alibi library to get the adult dataset with demographic features from a 1996 US census.

Note: Install alibi library for dataset dependencies and alibi_detect library for detector configuration from Pypi

```
!pip install alibi alibi_detect
```

```
import alibi
import matplotlib.pyplot as plt
import numpy as np
```

```
adult = alibi.datasets.fetch_adult()
X, y = adult.data, adult.target
```

(continues on next page)

(continued from previous page)

```
feature_names = adult.feature_names
category_map = adult.category_map
```

```
n_ref = 10000
n_test = 10000

X_ref, X_t0, X_t1 = X[:n_ref], X[n_ref:n_ref + n_test], X[n_ref + n_test:n_ref + 2 * n_
↪test]
categories_per_feature = {f: None for f in list(category_map.keys())}
```

Drift Detector Configuration

This example is based on the Categorical and mixed type data drift detection on income prediction tabular data from the alibi-detect documentation.

Creating detector and saving configuration

```
from alibi_detect.cd import TabularDrift
cd_tabular = TabularDrift(X_ref, p_val=.05, categories_per_feature=categories_per_
↪feature)
```

```
from alibi_detect.utils.saving import save_detector
filepath = "alibi-detector-artifacts"
save_detector(cd_tabular, filepath)
```

Detecting data drift directly

```
preds = cd_tabular.predict(X_t0,drift_type="feature")

labels = ['No!', 'Yes!']
print(f"Threshold {preds['data']['threshold']}")
for f in range(cd_tabular.n_features):
    fname = feature_names[f]
    is_drift = (preds['data']['p_val'][f] < preds['data']['threshold']).astype(int)
    stat_val, p_val = preds['data']['distance'][f], preds['data']['p_val'][f]
    print(f'{fname} -- Drift? {labels[is_drift]} -- Chi2 {stat_val:.3f} -- p-value {p_
↪val:.3f}')
```

```
Threshold 0.05
Age -- Drift? No! -- Chi2 0.012 -- p-value 0.508
Workclass -- Drift? No! -- Chi2 8.487 -- p-value 0.387
Education -- Drift? No! -- Chi2 4.753 -- p-value 0.576
Marital Status -- Drift? No! -- Chi2 3.160 -- p-value 0.368
Occupation -- Drift? No! -- Chi2 8.194 -- p-value 0.415
Relationship -- Drift? No! -- Chi2 0.485 -- p-value 0.993
Race -- Drift? No! -- Chi2 0.587 -- p-value 0.965
Sex -- Drift? No! -- Chi2 0.217 -- p-value 0.641
```

(continues on next page)

(continued from previous page)

```
Capital Gain -- Drift? No! -- Chi2 0.002 -- p-value 1.000
Capital Loss -- Drift? No! -- Chi2 0.002 -- p-value 1.000
Hours per week -- Drift? No! -- Chi2 0.012 -- p-value 0.508
Country -- Drift? No! -- Chi2 9.991 -- p-value 0.441
```

Serving

Now that we have the reference data and other configuration parameters, the next step will be to serve it using `mlserver`. For that, we will need to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

settings.json

```
%%writefile settings.json
{
    "debug": "true"
}
```

```
Overwriting settings.json
```

model-settings.json

```
%%writefile model-settings.json
{
  "name": "income-tabular-drift",
  "implementation": "mlserver_alibi_detect.AlibiDetectRuntime",
  "parameters": {
    "uri": "./alibi-detector-artifacts",
    "version": "v0.1.0",
    "extra": {
      "predict_parameters":{
        "drift_type": "feature"
      }
    }
  }
}
```

```
Overwriting model-settings.json
```

Start serving our model

Now that we have our config in-place, we can start the server by running mlserver start command. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our alibi-detect model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

inference_request = {
    "inputs": [
        {
            "name": "predict",
            "shape": X_t0.shape,
            "datatype": "FP32",
            "data": X_t0.tolist(),
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/income-tabular-drift/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)
```

View model response

```
import json
response_dict = json.loads(response.text)

labels = ['No!', 'Yes!']
for f in range(cd_tabular.n_features):
    stat = 'Chi2' if f in list(categories_per_feature.keys()) else 'K-S'
    fname = feature_names[f]
    is_drift = response_dict['outputs'][0]['data'][f]
    stat_val, p_val = response_dict['outputs'][1]['data'][f], response_dict['outputs
↪'][2]['data'][f]
    print(f'{fname} -- Drift? {labels[is_drift]} -- Chi2 {stat_val:.3f} -- p-value {p_
↪val:.3f}')
```

```
Age -- Drift? No! -- Chi2 0.012 -- p-value 0.508
Workclass -- Drift? No! -- Chi2 8.487 -- p-value 0.387
Education -- Drift? No! -- Chi2 4.753 -- p-value 0.576
```

(continues on next page)

(continued from previous page)

```
Marital Status -- Drift? No! -- Chi2 3.160 -- p-value 0.368
Occupation -- Drift? No! -- Chi2 8.194 -- p-value 0.415
Relationship -- Drift? No! -- Chi2 0.485 -- p-value 0.993
Race -- Drift? No! -- Chi2 0.587 -- p-value 0.965
Sex -- Drift? No! -- Chi2 0.217 -- p-value 0.641
Capital Gain -- Drift? No! -- Chi2 0.002 -- p-value 1.000
Capital Loss -- Drift? No! -- Chi2 0.002 -- p-value 1.000
Hours per week -- Drift? No! -- Chi2 0.012 -- p-value 0.508
Country -- Drift? No! -- Chi2 9.991 -- p-value 0.441
```

10.1.8 Serving HuggingFace Transformer Models

Out of the box, MLServer supports the deployment and serving of HuggingFace Transformer models with the following features:

- Loading of Transformer Model artifacts from the Hugging Face Hub.
- Model quantization & optimization using the Hugging Face Optimum library
- Request batching for GPU optimization (via adaptive batching and request batching)

In this example, we will showcase some of this features using an example model.

```
# Import required dependencies
import requests
```

Serving

Now that we have trained and serialised our model, we are ready to start serving it. For that, the initial step will be to set up a `model-settings.json` that instructs MLServer to load our artifact using the HuggingFace Inference Runtime.

We will show how to add share a task

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "parameters": {
        "extra": {
            "task": "text-generation",
            "pretrained_model": "distilgpt2"
        }
    }
}
```

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

```
inference_request = {
    "inputs": [
        {
          "name": "args",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["this is a test"],
        }
    ]
}

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request).json()
```

Using Optimum Optimized Models

We can also leverage the Optimum library that allows us to access quantized and optimized models.

We can download pretrained optimized models from the hub if available by enabling the `optimum_model` flag:

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "parameters": {
        "extra": {
            "task": "text-generation",
            "pretrained_model": "distilgpt2",
            "optimum_model": true
        }
    }
}
```

Once again, you are able to run the model using the MLServer CLI. As before this needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Send Test Request to Optimum Optimized Model

The request can now be sent using the same request structure but using optimized models for better performance.

```
inference_request = {
    "inputs": [
        {
          "name": "args",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["this is a test"],
```

(continues on next page)

(continued from previous page)

```
        }
    ]
}

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request).json()
```

Testing Supported Tasks

We can support multiple other transformers other than just text generation, below includes examples for a few other tasks supported.

Question Answering

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "parameters": {
        "extra": {
            "task": "question-answering"
        }
    }
}
```

Once again, you are able to run the model using the MLServer CLI.

```
mlserver start .
```

```
inference_request = {
    "inputs": [
        {
          "name": "question",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["what is your name?"],
        },
        {
          "name": "context",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["Hello, I am Seldon, how is it going"],
        }
    ]
}

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request).json()
```

Sentiment Analysis

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "parameters": {
        "extra": {
            "task": "text-classification"
        }
    }
}
```

Once again, you are able to run the model using the MLServer CLI.

```
mlserver start .
```

```
inference_request = {
    "inputs": [
        {
          "name": "args",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["This is terrible!"],
        }
    ]
}

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request).json()
```

GPU Acceleration

We can also evaluate GPU acceleration, we can test the speed on CPU vs GPU using the following parameters

Testing with CPU

We first test the time taken with the device=-1 which configures CPU by default

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "max_batch_size": 128,
    "max_batch_time": 1,
    "parameters": {
        "extra": {
            "task": "text-generation",
            "device": -1
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Once again, you are able to run the model using the MLServer CLI.

```
mlserver start .
```

```
inference_request = {
    "inputs": [
        {
          "name": "text_inputs",
          "shape": [1],
          "datatype": "BYTES",
          "data": ["This is a generation for the work" for i in range(512)],
        }
    ]
}

# Benchmark time
import time
start_time = time.monotonic()

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request)

print(f"Elapsed time: {time.monotonic() - start_time}")
```

We can see that it takes 81 seconds which is 8 times longer than the gpu example below.

Testing with GPU

IMPORTANT: Running the code below requries having a machine with GPU configured correctly to work for Tensorflow/Pytorch.

Now we'll run the benchmark with GPU configured, which we can do by setting `device=0`

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "parameters": {
        "extra": {
            "task": "text-generation",
            "device": 0
        }
    }
}
```

```
inference_request = {
    "inputs": [
        {
```

(continues on next page)

(continued from previous page)

```
            "name": "text_inputs",
            "shape": [1],
            "datatype": "BYTES",
            "data": ["This is a generation for the work" for i in range(512)],
        }
    ]
}

# Benchmark time
import time
start_time = time.monotonic()

requests.post("http://localhost:8080/v2/models/transformer/infer", json=inference_
↪request)

print(f"Elapsed time: {time.monotonic() - start_time}")
```

We can see that the elapsed time is 8 times less than the CPU version!

Adaptive Batching with GPU

We can also see how the adaptive batching capabilities can allow for GPU acceleration by grouping multiple incoming requests so they get processed in GPU batch.

In our case we can enable adaptive batching with the `max_batch_size` which in our case we will set it ot 128.

We will also configure `max_batch_time` which specifies` the maximum amount of time the MLServer orchestrator will wait before sending for inference.

```
%%writefile ./model-settings.json
{
    "name": "transformer",
    "implementation": "mlserver_huggingface.HuggingFaceRuntime",
    "max_batch_size": 128,
    "max_batch_time": 1,
    "parameters": {
        "extra": {
            "task": "text-generation",
            "pretrained_model": "distilgpt2",
            "device": 0
        }
    }
}
```

In order to achieve the throughput required of 50 requests per second, we will use the tool `vegeta` which performs load testing.

We can now see that we are able to see that the requests are batched and we receive 100% success eventhough the requests are sent one-by-one.

```
%%bash
jq -ncM '{"method": "POST", "header": {"Content-Type": ["application/json"] }, "url":
↪"http://localhost:8080/v2/models/transformer/infer", "body": "{\"inputs\":[{\"name\":\
```

(continues on next page)

(continued from previous page)

```
↪"text_inputs\",\"shape\":[1],\"datatype\":\"BYTES\",\"data\":[\"test\"]}]}" | @base64 }
↪' \
          | vegeta \
                -cpus="2" \
                attack \
                -duration="3s" \
                -rate="50" \
                -format=json \
          | vegeta \
                report \
                -type=text
```

10.2 MLServer Features

To see some of the advanced features included in MLServer (e.g. multi-model serving), check out the examples below.

- *Multi-Model Serving with multiple frameworks*
- *Loading / unloading models from a model repository*
- *Content-Type Decoding*
- *Custom Conda environment*
- *Serving custom models requiring JSON inputs or outputs*
- *Serving models through Kafka*

10.2.1 Multi-Model Serving

MLServer has been built with Multi-Model Serving (MMS) in mind. This means that, within a single instance of MLServer, you can serve multiple models under different paths. This also includes multiple versions of the same model.

This notebook shows an example of how you can leverage MMS with MLServer.

Training

We will first start by training 2 different models:

Name	Framework	Source	Trained Model Path
`mnist-svm`	`scikit-lea`	MNIST example from the `scikit-learn` documentation	`./models/mnist-svm/model.joblib`
`mushroom-xgbc`	`xgboost`	Mushrooms example from the `xgboost` Getting Started guide	`./models/mushroom-xgboost/model.json`

Training our `mnist-svm` model

```
# Original source code and more details can be found in:
# https://scikit-learn.org/stable/auto_examples/classification/plot_digits_
↪classification.html

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test_digits, y_train, y_test_digits = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)
```

```
import joblib
import os

mnist_svm_path = os.path.join("models", "mnist-svm")
os.makedirs(mnist_svm_path, exist_ok=True)

mnist_svm_model_path = os.path.join(mnist_svm_path, "model.joblib")
joblib.dump(classifier, mnist_svm_model_path)
```

Training our `mushroom-xgboost` model

```
# Original code and extra details can be found in:
# https://xgboost.readthedocs.io/en/latest/get_started.html#python

import os
import xgboost as xgb
import requests

from urllib.parse import urlparse
from sklearn.datasets import load_svmlight_file


TRAIN_DATASET_URL = 'https://raw.githubusercontent.com/dmlc/xgboost/master/demo/data/
```

(continues on next page)

(continued from previous page)

```
↪agaricus.txt.train'
TEST_DATASET_URL = 'https://raw.githubusercontent.com/dmlc/xgboost/master/demo/data/
↪agaricus.txt.test'


def _download_file(url: str) -> str:
    parsed = urlparse(url)
    file_name = os.path.basename(parsed.path)
    file_path = os.path.join(os.getcwd(), file_name)

    res = requests.get(url)

    with open(file_path, 'wb') as file:
        file.write(res.content)

    return file_path

train_dataset_path = _download_file(TRAIN_DATASET_URL)
test_dataset_path = _download_file(TEST_DATASET_URL)

# NOTE: Workaround to load SVMLight files from the XGBoost example
X_train, y_train = load_svmlight_file(train_dataset_path)
X_test_agar, y_test_agar = load_svmlight_file(test_dataset_path)
X_train = X_train.toarray()
X_test_agar = X_test_agar.toarray()

# read in data
dtrain = xgb.DMatrix(data=X_train, label=y_train)

# specify parameters via map
param = {'max_depth':2, 'eta':1, 'objective':'binary:logistic' }
num_round = 2
bst = xgb.train(param, dtrain, num_round)

bst
```

```
import os

mushroom_xgboost_path = os.path.join("models", "mushroom-xgboost")
os.makedirs(mushroom_xgboost_path, exist_ok=True)

mushroom_xgboost_model_path = os.path.join(mushroom_xgboost_path, "model.json")
bst.save_model(mushroom_xgboost_model_path)
```

Serving

The next step will be serving both our models within the same MLServer instance. For that, we will just need to create a `model-settings.json` file local to each of our models and a server-wide `settings.json`. That is,

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `models/mnist-svm/model-settings.json`: holds the configuration specific to our `mnist-svm` model (e.g. input type, runtime to use, etc.).
- `models/mushroom-xgboost/model-settings.json`: holds the configuration specific to our `mushroom-xgboost` model (e.g. input type, runtime to use, etc.).

settings.json

```
%%writefile settings.json
{
    "debug": "true"
}
```

models/mnist-svm/model-settings.json

```
%%writefile models/mnist-svm/model-settings.json
{
    "name": "mnist-svm",
    "implementation": "mlserver_sklearn.SKLearnModel",
    "parameters": {
        "version": "v0.1.0"
    }
}
```

models/mushroom-xgboost/model-settings.json

```
%%writefile models/mushroom-xgboost/model-settings.json
{
    "name": "mushroom-xgboost",
    "implementation": "mlserver_xgboost.XGBoostModel",
    "parameters": {
        "version": "v0.1.0"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Testing

By this point, we should have both our models getting served by MLServer. To make sure that everything is working as expected, let's send a request from each test set.

For that, we can use the Python types that the `mlserver` package provides out of box, or we can build our request manually.

Testing our `mnist-svm` model

```
import requests

x_0 = X_test_digits[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/mnist-svm/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

Testing our `mushroom-xgboost` model

```
import requests

x_0 = X_test_agar[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
```

(continues on next page)

(continued from previous page)

```
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/mushroom-xgboost/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

```

```

10.2.2 Model Repository API

MLServer supports loading and unloading models dynamically from a models repository. This allows you to enable and disable the models accessible by MLServer on demand. This extension builds on top of the support for *Multi-Model Serving*, letting you change at runtime which models is MLServer currently serving.

The API to manage the model repository is modelled after Triton's Model Repository extension to the V2 Dataplane and is thus fully compatible with it.

This notebook will walk you through an example using the Model Repository API.

Training

First of all, we will need to train some models. For that, we will re-use the models we trained previously in the *Multi-Model Serving example*. You can check the details on how they are trained following that notebook.

```
!cp -r ../mms/models/* ./models
```

Serving

Next up, we will start our `mlserver` inference server. Note that, by default, this will **load all our models**.

```
mlserver start .
```

List available models

Now that we've got our inference server up and running, and serving 2 different models, we can start using the Model Repository API. To get us started, we will first list all available models in the repository.

```
import requests

response = requests.post("http://localhost:8080/v2/repository/index", json={})
response.json()
```

As we can, the repository lists 2 models (i.e. `mushroom-xgboost` and `mnist-svm`). Note that the state for both is set to `READY`. This means that both models are loaded, and thus ready for inference.

Unloading our `mushroom-xgboost` model

We will now try to unload one of the 2 models, `mushroom-xgboost`. This will unload the model from the inference server but will keep it available on our model repository.

```
requests.post("http://localhost:8080/v2/repository/models/mushroom-xgboost/unload")
```

If we now try to list the models available in our repository, we will see that the `mushroom-xgboost` model is flagged as `UNAVAILABLE`. This means that it's present in the repository but it's not loaded for inference.

```
response = requests.post("http://localhost:8080/v2/repository/index", json={})
response.json()
```

Loading our `mushroom-xgboost` model back

We will now load our model back into our inference server.

```
requests.post("http://localhost:8080/v2/repository/models/mushroom-xgboost/load")
```

If we now try to list the models again, we will see that our `mushroom-xgboost` is back again, ready for inference.

```
response = requests.post("http://localhost:8080/v2/repository/index", json={})
response.json()
```

```

```

10.2.3 Content Type Decoding

MLServer extends the V2 inference protocol by adding support for a `content_type` annotation. This annotation can be provided either through the model metadata `parameters`, or through the input `parameters`. By leveraging the `content_type` annotation, we can provide the necessary information to MLServer so that it can *decode* the input payload from the "wire" V2 protocol to something meaningful to the model / user (e.g. a NumPy array).

This example will walk you through some examples which illustrate how this works, and how it can be extended.

Echo Inference Runtime

To start with, we will write a *dummy* runtime which just prints the input, the *decoded* input and returns it. This will serve as a testbed to showcase how the `content_type` support works.

Later on, we will extend this runtime by adding custom *codecs* that will decode our V2 payload to custom types.

```
%%writefile runtime.py
import json

from mlserver import MLModel
from mlserver.types import InferenceRequest, InferenceResponse, ResponseOutput
from mlserver.codecs import DecodedParameterName

_to_exclude = {
    "parameters": {DecodedParameterName, "headers"},
    'inputs': {"__all__": {"parameters": {DecodedParameterName, "headers"}}}
```

(continues on next page)

(continued from previous page)

```
}

class EchoRuntime(MLModel):
    async def predict(self, payload: InferenceRequest) -> InferenceResponse:
        outputs = []
        for request_input in payload.inputs:
            decoded_input = self.decode(request_input)
            print(f"------ Encoded Input ({request_input.name}) ------")
            as_dict = request_input.dict(exclude=_to_exclude)  # type: ignore
            print(json.dumps(as_dict, indent=2))
            print(f"------ Decoded input ({request_input.name}) ------")
            print(decoded_input)

            outputs.append(
                ResponseOutput(
                    name=request_input.name,
                    datatype=request_input.datatype,
                    shape=request_input.shape,
                    data=request_input.data
                )
            )

        return InferenceResponse(model_name=self.name, outputs=outputs)
```

As you can see above, this runtime will decode the incoming payloads by calling the `self.decode()` helper method. This method will check what's the right content type for each input in the following order:

1. Is there any content type defined in the `inputs[].parameters.content_type` field within the **request payload**?
2. Is there any content type defined in the `inputs[].parameters.content_type` field within the **model metadata**?
3. Is there any default content type that should be assumed?

Model Settings

In order to enable this runtime, we will also create a `model-settings.json` file. This file should be present (or accessible from) in the folder where we run `mlserver start .`.

```
%%writefile model-settings.json

{
    "name": "content-type-example",
    "implementation": "runtime.EchoRuntime"
}
```

Request Inputs

Our initial step will be to decide the content type based on the incoming `inputs[].parameters` field. For this, we will start our MLServer in the background (e.g. running `mlserver start .`)

```
import requests

payload = {
    "inputs": [
        {
            "name": "parameters-np",
            "datatype": "INT32",
            "shape": [2, 2],
            "data": [1, 2, 3, 4],
            "parameters": {
                "content_type": "np"
            }
        },
        {
            "name": "parameters-str",
            "datatype": "BYTES",
            "shape": [1],
            "data": "hello world ",
            "parameters": {
                "content_type": "str"
            }
        }
    ]
}

response = requests.post(
    "http://localhost:8080/v2/models/content-type-example/infer",
    json=payload
)
```

Codecs

As you've probably already noticed, writing request payloads compliant with both the V2 Inference Protocol requires a certain knowledge about both the V2 spec and the structure expected by each content type. To account for this and simplify usage, the MLServer package exposes a set of utilities which will help you interact with your models via the V2 protocol.

These helpers are mainly shaped as *"codecs"*. That is, abstractions which know how to *"encode"* and *"decode"* arbitrary Python datatypes to and from the V2 Inference Protocol.

Generally, we recommend using the existing set of codecs to generate your V2 payloads. This will ensure that requests and responses follow the right structure, and should provide a more seamless experience.

Following with our previous example, the same code could be rewritten using codecs as:

```
import requests
import numpy as np

from mlserver.types import InferenceRequest, InferenceResponse
```

(continues on next page)

(continued from previous page)

```
from mlserver.codecs import NumpyCodec, StringCodec

parameters_np = np.array([[1, 2], [3, 4]])
parameters_str = ["hello world "]

payload = InferenceRequest(
    inputs=[
        NumpyCodec.encode_input("parameters-np", parameters_np),
        # The `use_bytes=False` flag will ensure that the encoded payload is JSON-
↪compatible
        StringCodec.encode_input("parameters-str", parameters_str, use_bytes=False),
    ]
)

response = requests.post(
    "http://localhost:8080/v2/models/content-type-example/infer",
    json=payload.dict()
)

response_payload = InferenceResponse.parse_raw(response.text)
print(NumpyCodec.decode_output(response_payload.outputs[0]))
print(StringCodec.decode_output(response_payload.outputs[1]))
```

Note that the rewritten snippet now makes use of the built-in `InferenceRequest` class, which represents a V2 inference request. On top of that, it also uses the `NumpyCodec` and `StringCodec` implementations, which know how to encode a Numpy array and a list of strings into V2-compatible request inputs.

Model Metadata

Our next step will be to define the expected content type through the model metadata. This can be done by extending the `model-settings.json` file, and adding a section on inputs.

```
%%writefile model-settings.json

{
    "name": "content-type-example",
    "implementation": "runtime.EchoRuntime",
    "inputs": [
        {
            "name": "metadata-np",
            "datatype": "INT32",
            "shape": [2, 2],
            "parameters": {
                "content_type": "np"
            }
        },
        {
            "name": "metadata-str",
            "datatype": "BYTES",
            "shape": [11],
            "parameters": {
```

(continues on next page)

(continued from previous page)

```
                "content_type": "str"
            }
        }
    ]
}
```

After adding this metadata, we will re-start MLServer (e.g. `mlserver start .`) and we will send a new request without any explicit `parameters`.

```
import requests

payload = {
    "inputs": [
        {
            "name": "metadata-np",
            "datatype": "INT32",
            "shape": [2, 2],
            "data": [1, 2, 3, 4],
        },
        {
            "name": "metadata-str",
            "datatype": "BYTES",
            "shape": [11],
            "data": "hello world ",
        }
    ]
}

response = requests.post(
    "http://localhost:8080/v2/models/content-type-example/infer",
    json=payload
)
```

As you should be able to see in the server logs, MLServer will cross-reference the input names against the model metadata to find the right content type.

Custom Codecs

There may be cases where a custom inference runtime may need to encode / decode to custom datatypes. As an example, we can think of computer vision models which may only operate with `pillow` image objects.

In these scenarios, it's possible to extend the `Codec` interface to write our custom encoding logic. A `Codec`, is simply an object which defines a `decode()` and `encode()` methods. To illustrate how this would work, we will extend our custom runtime to add a custom `PillowCodec`.

```
%%writefile runtime.py
import io
import json

from PIL import Image

from mlserver import MLModel
```

(continues on next page)

(continued from previous page)

```
from mlserver.types import (
    InferenceRequest,
    InferenceResponse,
    RequestInput,
    ResponseOutput,
)
from mlserver.codecs import NumpyCodec, register_input_codec, DecodedParameterName
from mlserver.codecs.utils import InputOrOutput


_to_exclude = {
    "parameters": {DecodedParameterName},
    "inputs": {"__all__": {"parameters": {DecodedParameterName}}},
}


@register_input_codec
class PillowCodec(NumpyCodec):
    ContentType = "img"
    DefaultMode = "L"

    @classmethod
    def can_encode(cls, payload: Image) -> bool:
        return isinstance(payload, Image)

    @classmethod
    def _decode(cls, input_or_output: InputOrOutput) -> Image:
        if input_or_output.datatype != "BYTES":
            # If not bytes, assume it's an array
            image_array = super().decode_input(input_or_output)  # type: ignore
            return Image.fromarray(image_array, mode=cls.DefaultMode)

        encoded = input_or_output.data.__root__
        if isinstance(encoded, str):
            encoded = encoded.encode()

        return Image.frombytes(
            mode=cls.DefaultMode, size=input_or_output.shape, data=encoded
        )

    @classmethod
    def encode_output(cls, name: str, payload: Image) -> ResponseOutput:  # type: ignore
        byte_array = io.BytesIO()
        payload.save(byte_array, mode=cls.DefaultMode)

        return ResponseOutput(
            name=name, shape=payload.size, datatype="BYTES", data=byte_array.getvalue()
        )

    @classmethod
    def decode_output(cls, response_output: ResponseOutput) -> Image:
        return cls._decode(response_output)
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def encode_input(cls, name: str, payload: Image) -> RequestInput:  # type: ignore
        output = cls.encode_output(name, payload)
        return RequestInput(
            name=output.name,
            shape=output.shape,
            datatype=output.datatype,
            data=output.data,
        )

    @classmethod
    def decode_input(cls, request_input: RequestInput) -> Image:
        return cls._decode(request_input)


class EchoRuntime(MLModel):
    async def predict(self, payload: InferenceRequest) -> InferenceResponse:
        outputs = []
        for request_input in payload.inputs:
            decoded_input = self.decode(request_input)
            print(f"------ Encoded Input ({request_input.name}) ------")
            as_dict = request_input.dict(exclude=_to_exclude)  # type: ignore
            print(json.dumps(as_dict, indent=2))
            print(f"------ Decoded input ({request_input.name}) ------")
            print(decoded_input)

            outputs.append(
                ResponseOutput(
                    name=request_input.name,
                    datatype=request_input.datatype,
                    shape=request_input.shape,
                    data=request_input.data,
                )
            )

        return InferenceResponse(model_name=self.name, outputs=outputs)
```

We should now be able to restart our instance of MLServer (i.e. with the `mlserver start .` command), to send a few test requests.

```
import requests

payload = {
    "inputs": [
        {
            "name": "image-int32",
            "datatype": "INT32",
            "shape": [8, 8],
            "data": [
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0,
```

(continues on next page)

(continued from previous page)

```
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0,
                1, 0, 1, 0, 1, 0, 1, 0
            ],
            "parameters": {
                "content_type": "img"
            }
        },
        {
            "name": "image-bytes",
            "datatype": "BYTES",
            "shape": [8, 8],
            "data": (
                "10101010"
                "10101010"
                "10101010"
                "10101010"
                "10101010"
                "10101010"
                "10101010"
                "10101010"
            ),
            "parameters": {
                "content_type": "img"
            }
        }
    ]
}

response = requests.post(
    "http://localhost:8080/v2/models/content-type-example/infer",
    json=payload
)
```

As you should be able to see in the MLServer logs, the server is now able to decode the payload into a Pillow image. This example also illustrates how `Codec` objects can be compatible with multiple `datatype` values (e.g. tensor and `BYTES` in this case).

Request Codecs

So far, we've seen how you can specify codecs so that they get applied at the input level. However, it is also possible to use request-wide codecs that aggregate multiple inputs to decode the payload. This is usually relevant for cases where the models expect a multi-column input type, like a Pandas DataFrame.

To illustrate this, we will first tweak our `EchoRuntime` so that it prints the decoded contents at the request level.

```
%%writefile runtime.py
import json

```

(continues on next page)

(continued from previous page)

```
from mlserver import MLModel
from mlserver.types import InferenceRequest, InferenceResponse, ResponseOutput
from mlserver.codecs import DecodedParameterName

_to_exclude = {
    "parameters": {DecodedParameterName},
    'inputs': {"__all__": {"parameters": {DecodedParameterName}}}
}

class EchoRuntime(MLModel):
    async def predict(self, payload: InferenceRequest) -> InferenceResponse:
        print("------ Encoded Input (request) ------")
        as_dict = payload.dict(exclude=_to_exclude)  # type: ignore
        print(json.dumps(as_dict, indent=2))
        print("------ Decoded input (request) ------")
        decoded_request = None
        if payload.parameters:
            decoded_request = getattr(payload.parameters, DecodedParameterName)
        print(decoded_request)

        outputs = []
        for request_input in payload.inputs:
            outputs.append(
                ResponseOutput(
                    name=request_input.name,
                    datatype=request_input.datatype,
                    shape=request_input.shape,
                    data=request_input.data
                )
            )

        return InferenceResponse(model_name=self.name, outputs=outputs)
```

We should now be able to restart our instance of MLServer (i.e. with the `mlserver start .` command), to send a few test requests.

```
import requests

payload = {
    "inputs": [
        {
            "name": "parameters-np",
            "datatype": "INT32",
            "shape": [2, 2],
            "data": [1, 2, 3, 4],
            "parameters": {
                "content_type": "np"
            }
        },
        {
            "name": "parameters-str",
```

(continues on next page)

(continued from previous page)

```
            "datatype": "BYTES",
            "shape": [2, 11],
            "data": ["hello world ", "bye bye "],
            "parameters": {
                "content_type": "str"
            }
        }
    ],
    "parameters": {
        "content_type": "pd"
    }
}

response = requests.post(
    "http://localhost:8080/v2/models/content-type-example/infer",
    json=payload
)
```

10.2.4 Custom Conda environments in MLServer

It's not unusual that model runtimes require extra dependencies that are not direct dependencies of MLServer. This is the case when we want to use *custom runtimes*, but also when our model artifacts are the output of older versions of a toolkit (e.g. models trained with an older version of SKLearn).

In these cases, since these dependencies (or dependency versions) are not known in advance by MLServer, they **won't be included in the default `seldonio/mlserver` Docker image**. To cover these cases, the **`seldonio/mlserver` Docker image allows you to load custom environments** before starting the server itself.

This example will walk you through how to create and save an custom environment, so that it can be loaded in MLServer without any extra change to the `seldonio/mlserver` Docker image.

Define our environment

For this example, we will create a custom environment to serve a model trained with an older version of Scikit-Learn. The first step will be define this environment, using a `environment.yml`.

Note that these environments can also be created on the fly as we go, and then serialised later.

```
%%writefile environment.yml

name: old-sklearn
channels:
    - conda-forge
dependencies:
    - python == 3.8
    - scikit-learn == 0.24.2
    - joblib == 0.17.0
    - requests
    - pip
```

(continues on next page)

(continued from previous page)

```
    - pip:
        - mlserver == 1.1.0
        - mlserver-sklearn == 1.1.0
```

Train model in our custom environment

To illustrate the point, we will train a Scikit-Learn model using our older environment.

The first step will be to create and activate an environment which reflects what's outlined in our `environment.yml` file.

> **NOTE:** If you are running this from a Jupyter Notebook, you will need to restart your Jupyter instance so that it runs from this environment.

```
!conda env create --force -f environment.yml
!conda activate old-sklearn
```

We can now train and save a Scikit-Learn model using the older version of our environment. This model will be serialised as `model.joblib`.

You can find more details of this process in the *Scikit-Learn example*.

```
# Original source code and more details can be found in:
# https://scikit-learn.org/stable/auto_examples/classification/plot_digits_
↪classification.html

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)
```

```
import joblib

model_file_name = "model.joblib"
joblib.dump(classifier, model_file_name)
```

Serialise our custom environment

Lastly, we will need to serialise our environment in the format expected by MLServer. To do that, we will use a tool called `conda-pack`.

This tool, will save a portable version of our environment as a `.tar.gz` file, also known as *tarball*.

```
!conda pack --force -n old-sklearn -o old-sklearn.tar.gz
```

Serving

Now that we have defined our environment (and we've got a sample artifact trained in that environment), we can move to serving our model.

To do that, we will first need to select the right runtime through a `model-settings.json` config file.

```
%%writefile model-settings.json
{
    "name": "mnist-svm",
    "implementation": "mlserver_sklearn.SKLearnModel"
}
```

We can then spin up our model, using our custom environment, leveraging MLServer's Docker image. Keep in mind that **you will need Docker installed in your machine to run this example**.

Our Docker command will need to take into account the following points:

- Mount the example's folder as a volume so that it can be accessed from within the container.
- Let MLServer know that our custom environment's tarball can be found as `old-sklearn.tar.gz`.
- Expose port `8080` so that we can send requests from the outside.

From the command line, this can be done using Docker's CLI as:

```
docker run -it --rm \
    -v "$PWD":/mnt/models \
    -e "MLSERVER_ENV_TARBALL=/mnt/models/old-sklearn.tar.gz" \
    -p 8080:8080 \
    seldonio/mlserver:1.1.0-slim
```

Note that we need to keep the server running in the background while we send requests. Therefore, it's best to run this command on a separate terminal session.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = X_test[0:1]
inference_request = {
```

(continues on next page)

(continued from previous page)

```
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/mnist-svm/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

```
```

10.2.5 Serving a custom model with JSON serialization

The `mlserver` package comes with inference runtime implementations for `scikit-learn` and `xgboost` models. However, some times we may also need to roll out our own inference server, with custom logic to perform inference. To support this scenario, MLServer makes it really easy to create your own extensions, which can then be containerised and deployed in a production environment.

Overview

In this example, we create a simple `Hello World JSON` model that parses and modifies a JSON data chunk. This is often useful as a means to quickly bootstrap existing models that utilize JSON based model inputs.

Serving

The next step will be to serve our model using `mlserver`. For that, we will first implement an extension which serve as the *runtime* to perform inference using our custom `Hello World JSON` model.

Custom inference runtime

This is a trivial model to demonstrate how to conceptually work with JSON inputs / outputs. In this example:

- Parse the JSON input from the client
- Create a JSON response echoing back the client request as well as a server generated message

```
%%writefile jsonmodels.py
import json

from typing import Dict, Any
from mlserver import MLModel, types
from mlserver.codecs import StringCodec
```

(continues on next page)

(continued from previous page)

```
class JsonHelloWorldModel(MLModel):
    async def load(self) -> bool:
        # Perform additional custom initialization here.
        print("Initialize model")

        # Set readiness flag for model
        return await super().load()

    async def predict(self, payload: types.InferenceRequest) -> types.InferenceResponse:
        request = self._extract_json(payload)
        response = {
            "request": request,
            "server_response": "Got your request. Hello from the server.",
        }
        response_bytes = json.dumps(response).encode("UTF-8")

        return types.InferenceResponse(
            id=payload.id,
            model_name=self.name,
            model_version=self.version,
            outputs=[
                types.ResponseOutput(
                    name="echo_response",
                    shape=[len(response_bytes)],
                    datatype="BYTES",
                    data=[response_bytes],
                    parameters=types.Parameters(content_type="str"),
                )
            ],
        )

    def _extract_json(self, payload: types.InferenceRequest) -> Dict[str, Any]:
        inputs = {}
        for inp in payload.inputs:
            inputs[inp.name] = json.loads(
                "".join(self.decode(inp, default_codec=StringCodec))
            )

        return inputs
```

Settings files

The next step will be to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

settings.json

```
%%writefile settings.json
{
    "debug": "true"
}
```

model-settings.json

```
%%writefile model-settings.json
{
    "name": "json-hello-world",
    "implementation": "jsonmodels.JsonHelloWorldModel"
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request (REST)

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests
import json
from mlserver.types import InferenceResponse
from mlserver.codecs.string import StringRequestCodec
from pprint import PrettyPrinter

pp = PrettyPrinter(indent=1)

inputs = {"name": "Foo Bar", "message": "Hello from Client (REST)!"}
```

(continues on next page)

(continued from previous page)

```
# NOTE: this uses characters rather than encoded bytes. It is recommended that you use␣
↪the `mlserver` types to assist in the correct encoding.
inputs_string = json.dumps(inputs)

inference_request = {
    "inputs": [
        {
            "name": "echo_request",
            "shape": [len(inputs_string)],
            "datatype": "BYTES",
            "data": [inputs_string],
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/json-hello-world/infer"
response = requests.post(endpoint, json=inference_request)

print(f"full response:\n")
print(response)
# retrive text output as dictionary
inference_response = InferenceResponse.parse_raw(response.text)
raw_json = StringRequestCodec.decode_response(inference_response)
output = json.loads(raw_json[0])
print(f"\ndata part:\n")
pp.pprint(output)
```

Send test inference request (gRPC)

Utilizing string data with the gRPC interface can be a bit tricky. To ensure we are correctly handling inputs and outputs we will be handled correctly.

For simplicity in this case, we leverage the Python types that `mlserver` provides out of the box. Alternatively, the gRPC stubs can be generated regenerated from the V2 specification directly for use by non-Python as well as Python clients.

```
import requests
import json
import grpc
from mlserver.codecs.string import StringRequestCodec
import mlserver.grpc.converters as converters
import mlserver.grpc.dataplane_pb2_grpc as dataplane
import mlserver.types as types
from pprint import PrettyPrinter

pp = PrettyPrinter(indent=1)

model_name = "json-hello-world"
inputs = {"name": "Foo Bar", "message": "Hello from Client (gRPC)!"}
inputs_bytes = json.dumps(inputs).encode("UTF-8")
```

(continues on next page)

(continued from previous page)

```
inference_request = types.InferenceRequest(
    inputs=[
        types.RequestInput(
            name="echo_request",
            shape=[len(inputs_bytes)],
            datatype="BYTES",
            data=[inputs_bytes],
            parameters=types.Parameters(content_type="str"),
        )
    ]
)

inference_request_g = converters.ModelInferRequestConverter.from_types(
    inference_request, model_name=model_name, model_version=None
)

grpc_channel = grpc.insecure_channel("localhost:8081")
grpc_stub = dataplane.GRPCInferenceServiceStub(grpc_channel)

response = grpc_stub.ModelInfer(inference_request_g)

print(f"full response:\n")
print(response)
# retrive text output as dictionary
inference_response = converters.ModelInferResponseConverter.to_types(response)
raw_json = StringRequestCodec.decode_response(inference_response)
output = json.loads(raw_json[0])
print(f"\ndata part:\n")
pp.pprint(output)
```

10.2.6 Serving models through Kafka

Out of the box, MLServer provides support to receive inference requests from Kafka. The Kafka server can run side-by-side with the REST and gRPC ones, and adds a new interface to interact with your model. The inference responses coming back from your model, will also get written back to their own output topic.

In this example, we will showcase the integration with Kafka by serving a *Scikit-Learn* model thorugh Kafka.

Run Kafka

We are going to start by running a simple local docker deployment of kafka that we can test against. This will be a minimal cluster that will consist of a single zookeeper node and a single broker.

You need to have Java installed in order for it to work correctly.

```
!wget https://apache.mirrors.nublue.co.uk/kafka/2.8.0/kafka_2.12-2.8.0.tgz
!tar -zxvf kafka_2.12-2.8.0.tgz
!./kafka_2.12-2.8.0/bin/kafka-storage.sh format -t OXn8RTSlQdmxwjhKnSB_6A -c ./kafka_2.
↪12-2.8.0/config/kraft/server.properties
```

Run the no-zookeeper kafka broker

Now you can just run it with the following command outside the terminal:

```
!./kafka_2.12-2.8.0/bin/kafka-server-start.sh ./kafka_2.12-2.8.0/config/kraft/server.
↪properties
```

Create Topics

Now we can create the input and output topics required

```
!./kafka_2.12-2.8.0/bin/kafka-topics.sh --create --topic mlserver-input --partitions 1 --
↪replication-factor 1 --bootstrap-server localhost:9092
!./kafka_2.12-2.8.0/bin/kafka-topics.sh --create --topic mlserver-output --partitions 1 -
↪-replication-factor 1 --bootstrap-server localhost:9092
```

Training

The first step will be to train a simple `scikit-learn` model. For that, we will use the MNIST example from the `scikit-learn` documentation which trains an SVM model.

```
# Original source code and more details can be found in:
# https://scikit-learn.org/stable/auto_examples/classification/plot_digits_
↪classification.html

# Import datasets, classifiers and performance metrics
from sklearn import datasets, svm, metrics
from sklearn.model_selection import train_test_split

# The digits dataset
digits = datasets.load_digits()

# To apply a classifier on this data, we need to flatten the image, to
# turn the data in a (samples, feature) matrix:
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# Split data into train and test subsets
X_train, X_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False)

# We learn the digits on the first half of the digits
classifier.fit(X_train, y_train)
```

Saving our trained model

To save our trained model, we will serialise it using `joblib`. While this is not a perfect approach, it's currently the recommended method to persist models to disk in the `scikit-learn` documentation.

Our model will be persisted as a file named `mnist-svm.joblib`

```
import joblib

model_file_name = "mnist-svm.joblib"
joblib.dump(classifier, model_file_name)
```

Serving

Now that we have trained and saved our model, the next step will be to serve it using `mlserver`. For that, we will need to create 2 configuration files:

- `settings.json`: holds the configuration of our server (e.g. ports, log level, etc.).
- `model-settings.json`: holds the configuration of our model (e.g. input type, runtime to use, etc.).

Note that, the `settings.json` file will contain our Kafka configuration, including the address of the Kafka broker and the input / output topics that will be used for inference.

settings.json

```
%%writefile settings.json
{
    "debug": "true",
    "kafka_enabled": "true"
}
```

model-settings.json

```
%%writefile model-settings.json
{
    "name": "mnist-svm",
    "implementation": "mlserver_sklearn.SKLearnModel",
    "parameters": {
        "uri": "./mnist-svm.joblib",
        "version": "v0.1.0"
    }
}
```

Start serving our model

Now that we have our config in-place, we can start the server by running `mlserver start .`. This needs to either be ran from the same directory where our config files are or pointing to the folder where they are.

```
mlserver start .
```

Since this command will start the server and block the terminal, waiting for requests, this will need to be ran in the background on a separate terminal.

Send test inference request

We now have our model being served by `mlserver`. To make sure that everything is working as expected, let's send a request from our test set.

For that, we can use the Python types that `mlserver` provides out of box, or we can build our request manually.

```
import requests

x_0 = X_test[0:1]
inference_request = {
    "inputs": [
        {
          "name": "predict",
          "shape": x_0.shape,
          "datatype": "FP32",
          "data": x_0.tolist()
        }
    ]
}

endpoint = "http://localhost:8080/v2/models/mnist-svm/versions/v0.1.0/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

Send inference request through Kafka

Now that we have verified that our server is accepting REST requests, we will try to send a new inference request through Kafka. For this, we just need to send a request to the `mlserver-input` topic (which is the default input topic):

```
import json
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers="localhost:9092")

headers = {
    "mlserver-model": b"mnist-svm",
    "mlserver-version": b"v0.1.0",
}
```

(continues on next page)

(continued from previous page)

```
producer.send(
    "mlserver-input",
    json.dumps(inference_request).encode("utf-8"),
    headers=list(headers.items()))
```

Once the message has gone into the queue, the Kafka server running within MLServer should receive this message and run inference. The prediction output should then get posted into an output queue, which will be named `mlserver-output` by default.

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    "mlserver-output",
    bootstrap_servers="localhost:9092",
    auto_offset_reset="earliest")

for msg in consumer:
    print(f"key: {msg.key}")
    print(f"value: {msg.value}\n")
    break
```

As we should now be able to see above, the results of our inference request should now be visible in the output Kafka queue.

```

```

10.3 Tutorials

Tutorials are designed to be *beginner-friendly* and walk through accomplishing a series of tasks using MLServer (and other tools).

- *Deploying a Custom Tensorflow Model with MLServer and Seldon Core*

10.3.1 Deploying a Custom Tensorflow Model with MLServer and Seldon Core

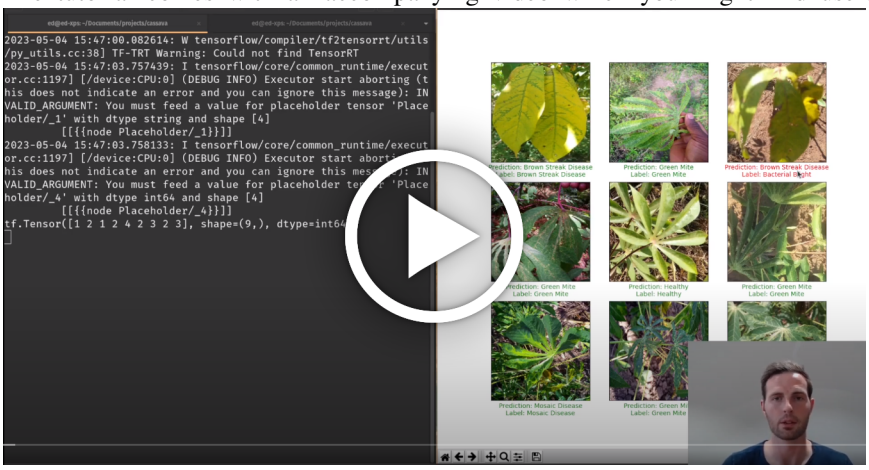
Background

Intro

This tutorial walks through the steps required to take a python ML model from your machine to a production deployment on Kubernetes. More specifically we'll cover:

- Running the model locally
- Turning the ML model into an API
- Containerizing the model
- Storing the container in a registry
- Deploying the model to Kubernetes (with Seldon Core)
- Scaling the model

The tutorial comes with an accompanying video which you might find useful as you work through the steps:



The slides used in the video can be found here.

The Use Case

For this tutorial, we're going to use the Cassava dataset available from the Tensorflow Catalog. This dataset includes leaf images from the cassava plant. Each plant can be classified as either "healthly" or as having one of four diseases (Mosaic Disease, Bacterial Blight, Green Mite, Brown Streak Disease).

We won't go through the steps of training the classifier. Instead, we'll be using a pre-trained one available on TensorFlow Hub. You can find the model details here.

Getting Set Up

The easiest way to run this example is to clone the repository located here:

```
git clone https://github.com/SeldonIO/cassava-example.git
```

If you've already cloned the MLServer repository, you can also find it in `docs/examples/cassava`.

Once you've done that, you can just run:

```
cd cassava-example/
```

```
pip install -r requirements.txt
```

And it'll set you up with all the libraries required to run the code.

Running The Python App

The starting point for this tutorial is python script `app.py`. This is typical of the kind of python code we'd run standalone or in a jupyter notebook. Let's familiarise ourself with the code:

```
from helpers import plot, preprocess
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_hub as hub

# Fixes an issue with Jax and TF competing for GPU
tf.config.experimental.set_visible_devices([], 'GPU')

# Load the model
model_path = './model'
classifier = hub.KerasLayer(model_path)

# Load the dataset and store the class names
dataset, info = tfds.load('cassava', with_info=True)
class_names = info.features['label'].names + ['unknown']

# Select a batch of examples and plot them
batch_size = 9
batch = dataset['validation'].map(preprocess).batch(batch_size).as_numpy_iterator()
examples = next(batch)
plot(examples, class_names)

# Generate predictions for the batch and plot them against their labels
predictions = classifier(examples['image'])
predictions_max = tf.argmax(predictions, axis=-1)
print(predictions_max)
plot(examples, class_names, predictions_max)
```

First up, we're importing a couple of functions from our `helpers.py` file:

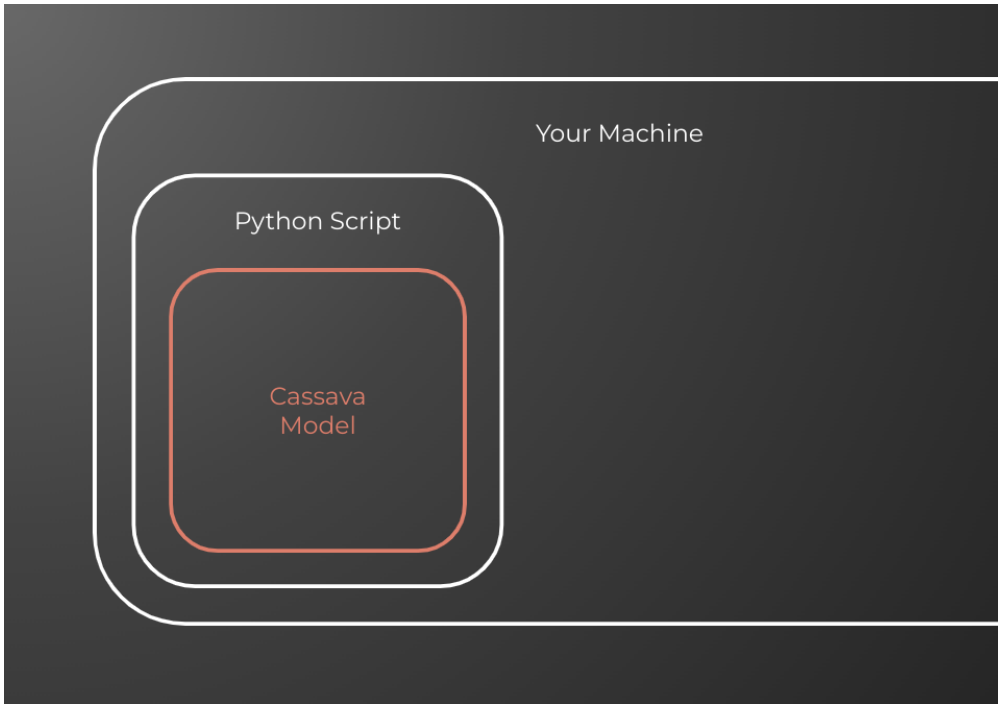
- `plot` provides the visualisation of the samples, labels and predictions.
- `preprocess` is used to resize images to 224x224 pixels and normalize the RGB values.

The rest of the code is fairly self-explanatory from the comments. We load the model and dataset, select some examples, make predictions and then plot the results.

Try it yourself by running:

```
python app.py
```

Here's what our setup currently looks like:



Creating an API for The Model

The problem with running our code like we did earlier is that it's not accessible to anyone who doesn't have the python script (and all of it's dependencies). A good way to solve this is to turn our model into an API.

Typically people turn to popular python web servers like Flask or FastAPI. This is a good approach and gives us lots of flexibility but it also requires us to do a lot of the work ourselves. We need to impelement routes, set up logging, capture metrics and define an API schema among other things. A simpler way to tackle this problem is to use an inference server. For this tutorial we're going to use the open source MLServer framework.

MLServer supports a bunch of inference runtimes out of the box, but it also supports custom python code which is what we'll use for our Tensorflow model.

Setting Things Up

In order to get our model ready to run on MLServer we need to wrap it in a single python class with two methods, `load()` and `predict()`. Let's take a look at the code (found in `model/serve-model.py`):

```
from mlserver import MLModel
from mlserver.codecs import decode_args
import numpy as np
import tensorflow as tf
import tensorflow_hub as hub
```

(continues on next page)

(continued from previous page)

```
# Define a class for our Model, inheriting the MLModel class from MLServer
class CassavaModel(MLModel):

  # Load the model into memory
  async def load(self) -> bool:
    tf.config.experimental.set_visible_devices([], 'GPU')
    model_path = '.'
    self._model = hub.KerasLayer(model_path)
    self.ready = True
    return self.ready

  # Logic for making predictions against our model
  @decode_args
  async def predict(self, payload: np.ndarray) -> np.ndarray:
    # convert payload to tf.tensor
    payload_tensor = tf.constant(payload)

    # Make predictions
    predictions = self._model(payload_tensor)
    predictions_max = tf.argmax(predictions, axis=-1)

    # convert predictions to np.ndarray
    response_data = np.array(predictions_max)

    return response_data
```

The `load()` method is used to define any logic required to set up our model for inference. In our case, we're loading the model weights into `self._model`. The `predict()` method is where we include all of our prediction logic.

You may notice that we've slightly modified our code from earlier (in `app.py`). The biggest change is that it is now wrapped in a single class `CassavaModel`.

The only other task we need to do to run our model on MLServer is to specify a `model-settings.json` file:

```
{
    "name": "cassava",
    "implementation": "serve-model.CassavaModel"
}
```

This is a simple configuration file that tells MLServer how to handle our model. In our case, we've provided a name for our model and told MLServer where to look for our model class (`serve-model.CassavaModel`).

Serving The Model

We're now ready to serve our model with MLServer. To do that we can simply run:

```
mlserver start model/
```

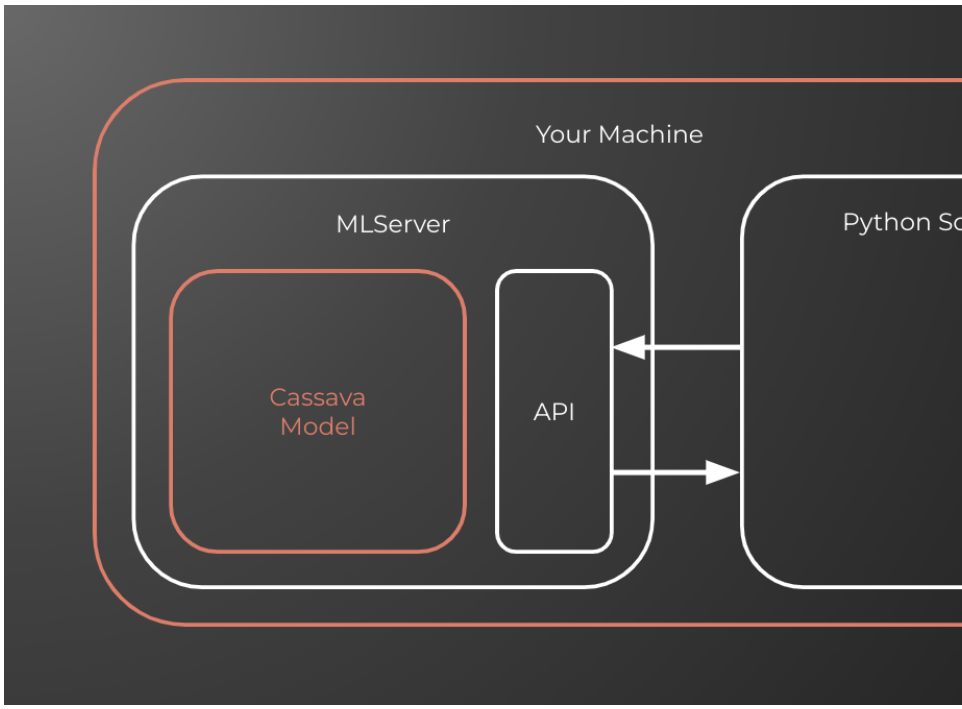
MLServer will now start up, load our cassava model and provide access through both a REST and gRPC API.

Making Predictions Using The API

Now that our API is up and running. Open a new terminal window and navigate back to the root of this repository. We can then send predictions to our api using the `test.py` file by running:

```
python test.py --local
```

Our setup has now evloved and looks like this:



Containerizing The Model

Containers are an easy way to package our application together with it's runtime and dependencies. More importantly, containerizing our model allows it to run in a variety of different environments.

> **Note:** you will need Docker installed to run this section of the tutorial. You'll also need a docker hub account or another container registry.

Taking our model and packaging it into a container manually can be a pretty tricky process and requires knowledge of writing Dockerfiles. Thankfully MLServer removes this complexity and provides us with a simple `build` command.

Before we run this command, we need to provide our dependencies in either a `requirements.txt` or a `conda.env` file. The requirements file we'll use for this example is stored in `model/requirements.txt`:

```
tensorflow==2.12.0
tensorflow-hub==0.13.0
```

> Notice that we didn't need to include `mlserver` in our requirements? That's because the builder image has mlserver included already.

We're now ready to build our container image using:

```
mlserver build model/ -t [YOUR_CONTAINER_REGISTRY]/[IMAGE_NAME]
```

Make sure you replace `YOUR_CONTAINER_REGISTRY` and `IMAGE_NAME` with your dockerhub username and a suitable name e.g. "bobsmith/cassava".

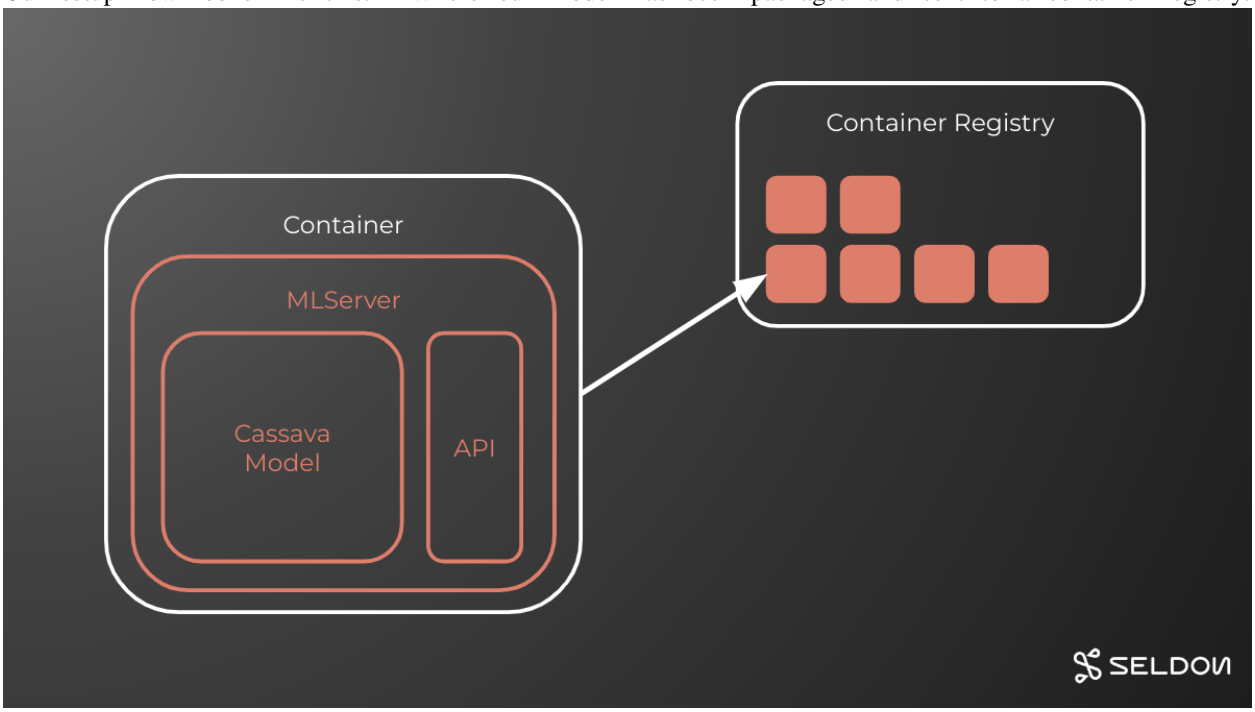
MLServer will now build the model into a container image for us. We can check the output of this by running:

```
docker images
```

Finally, we want to send this container image to be stored in our container registry. We can do this by running:

```
docker push [YOUR_CONTAINER_REGISTRY]/[IMAGE_NAME]
```

Our setup now looks like this. Where our model has been packaged and sent to a container registry:



Deploying to Kubernetes

Now that we've turned our model into a production-ready API, containerized it and pushed it to a registry, it's time to deploy our model.

We're going to use a popular open source framework called Seldon Core to deploy our model. Seldon Core is great because it combines all of the awesome cloud-native features we get from Kubernetes but it also adds machine-learning specific features.

This tutorial assumes you already have a Seldon Core cluster up and running. If that's not the case, head over the installation instructions and get set up first. You'll also need to install the `kubectl` *command line interface.*

Creating the Deployment

To create our deployment with Seldon Core we need to create a small configuration file that looks like this:

You can find this file named `deployment.yaml` *in the base folder of this tutorial's repository.*

```
apiVersion: machinelearning.seldon.io/v1
kind: SeldonDeployment
metadata:
  name: cassava
spec:
  protocol: v2
  predictors:
    - componentSpecs:
        - spec:
            containers:
              - image: YOUR_CONTAINER_REGISTRY/IMAGE_NAME
                name: cassava
                imagePullPolicy: Always
      graph:
        name: cassava
        type: MODEL
      name: cassava
```

Make sure you replace `YOUR_CONTAINER_REGISTRY` and `IMAGE_NAME` with your dockerhub username and a suitable name e.g. "bobsmith/cassava".

We can apply this configuration file to our Kubernetes cluster just like we would for any other Kubernetes object using:

```
kubectl create -f deployment.yaml
```

To check our deployment is up and running we can run:

```
kubectl get pods
```

We should see `STATUS = Running` once our deployment has finalized.

Testing the Deployment

Now that our model is up and running on a Kubernetes cluster (via Seldon Core), we can send some test inference requests to make sure it's working.

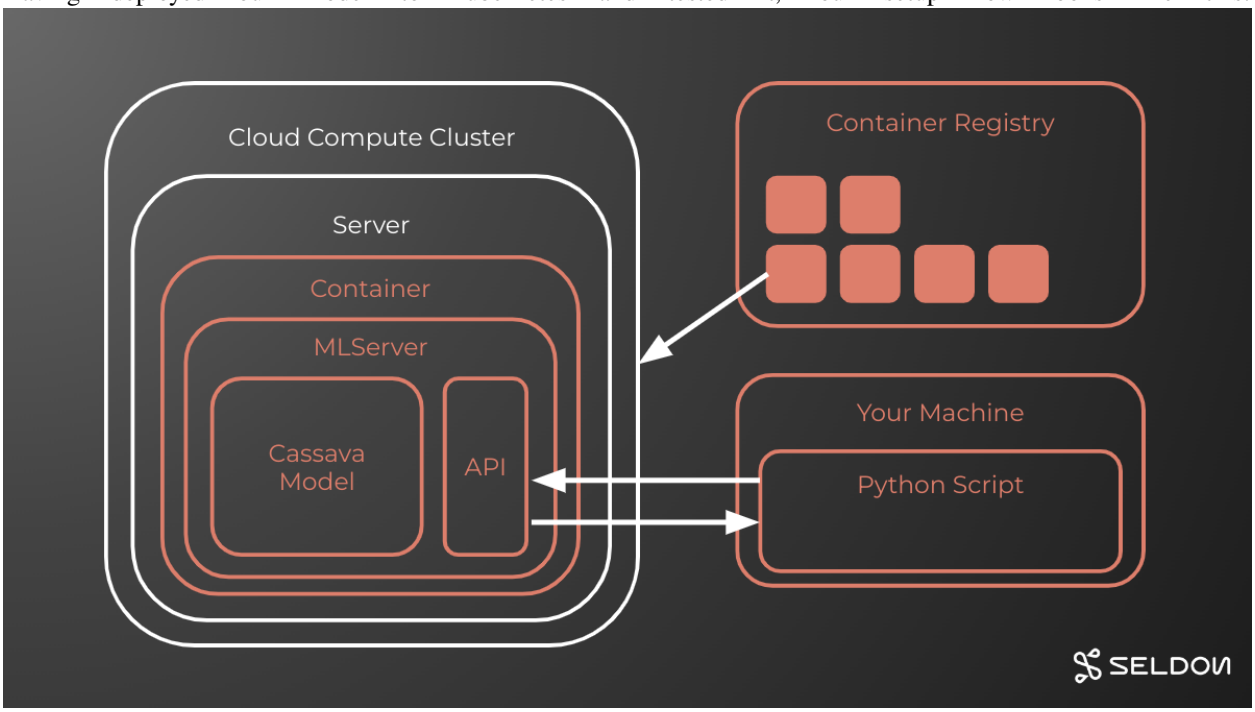
To do this, we simply run the `test.py` file in the following way:

```
python test.py --remote
```

This script will randomly select some test samples, send them to the cluster, gather the predictions and then plot them for us.

A note on running this yourself: *This example is set up to connect to a kubernetes cluster running locally on your machine. If yours is local too, you'll need to make sure you port forward before sending requests. If your cluster is remote, you'll need to change the* `inference_url` *variable on line 21 of* `test.py`.

Having deployed our model to kubernetes and tested it, our setup now looks like this:



Scaling the Model

Our model is now running in a production environment and able to handle requests from external sources. This is awesome but what happens as the number of requests being sent to our model starts to increase? Eventually, we'll reach the limit of what a single server can handle. Thankfully, we can get around this problem by scaling our model horizontally.

Kubernetes and Seldon Core make this really easy to do by simply running:

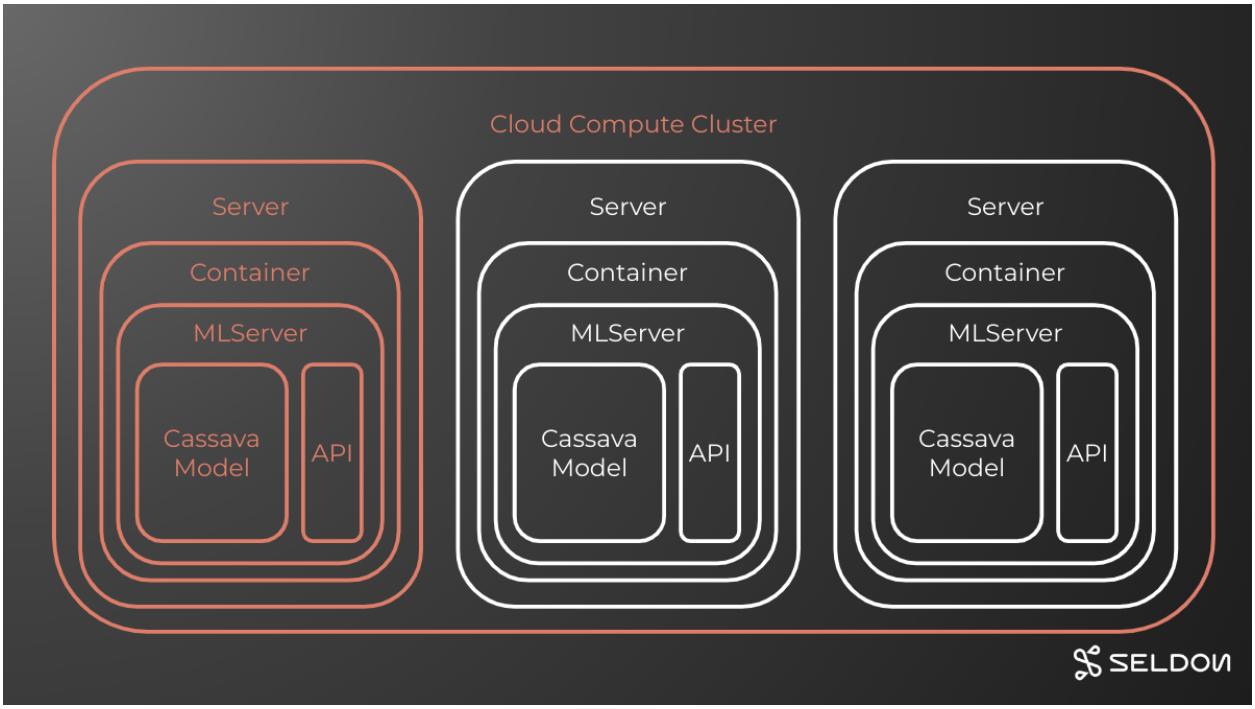
```
kubectl scale sdep cassava --replicas=3
```

We can replace the `--replicas=3` with any number we want to scale to.

To watch the servers scaling out we can run:

```
kubectl get pods --watch
```

Once the new replicas have finished rolling out, our setup now looks like this:



In this tutorial we've scaled the model out manually to show how it works. In a real environment we'd want to set up auto-scaling to make sure our prediction API is always online and performing as expected.

CHAPTER

ELEVEN

CHANGELOG

11.1 1.3.5 - 10 Jul 2023

11.1.1 What's Changed

- Rename HF codec to `hf` by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1268
- Publish is_drift metric to Prom by @joshsgoldstein in https://github.com/SeldonIO/MLServer/pull/1263

11.1.2 New Contributors

- @joshsgoldstein made their first contribution in https://github.com/SeldonIO/MLServer/pull/1263

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.3.4...1.3.5

Changes

11.2 1.3.4 - 21 Jun 2023

11.2.1 What's Changed

- Silent logging by @dtpryce in https://github.com/SeldonIO/MLServer/pull/1230
- Fix `mlserver infer` with `BYTES` by @RafalSkolasinski in https://github.com/SeldonIO/MLServer/pull/1213

11.2.2 New Contributors

- @dtpryce made their first contribution in https://github.com/SeldonIO/MLServer/pull/1230

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.3.3...1.3.4

Changes

11.3 1.3.3 - 05 Jun 2023

11.3.1 What's Changed

- Add default LD_LIBRARY_PATH env var by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1120
- Adding cassava tutorial (mlserver + seldon core) by @edshee in https://github.com/SeldonIO/MLServer/pull/1156
- Add docs around converting to / from JSON by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1165
- Document SKLearn available outputs by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1167
- Fix minor typo in `alibi-explain` tests by @ascillitoe in https://github.com/SeldonIO/MLServer/pull/1170
- Add support for `.ubj` models and improve XGBoost docs by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1168
- Fix content type annotations for pandas codecs by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1162
- Added option to configure the grpc histogram by @cristiancl25 in https://github.com/SeldonIO/MLServer/pull/1143
- Add OS classifiers to project's metadata by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1171
- Don't use `qsize` for parallel worker queue by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1169
- Fix small typo in Python API docs by @krishanbhasin-gc in https://github.com/SeldonIO/MLServer/pull/1174
- Fix star import in `mlserver.codecs.*` by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1172

11.3.2 New Contributors

- @cristiancl25 made their first contribution in https://github.com/SeldonIO/MLServer/pull/1143
- @krishanbhasin-gc made their first contribution in https://github.com/SeldonIO/MLServer/pull/1174

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.3.2...1.3.3

Changes

11.4 1.3.2 - 10 May 2023

11.4.1 What's Changed

- Use default initialiser if not using a custom env by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1104
- Add support for online drift detectors by @ascillitoe in https://github.com/SeldonIO/MLServer/pull/1108
- added intera and inter op parallelism parameters to the hugggingface ... by @saeid93 in https://github.com/SeldonIO/MLServer/pull/1081
- Fix settings reference in runtime docs by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1109
- Bump Alibi libs requirements by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1121
- Add default LD_LIBRARY_PATH env var by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1120
- Ignore both .metrics and .envs folders by @adriangonz in https://github.com/SeldonIO/MLServer/pull/1132

11.4.2 New Contributors

- @ascillitoe made their first contribution in https://github.com/SeldonIO/MLServer/pull/1108

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.3.1...1.3.2

Changes

11.5 1.3.1 - 27 Apr 2023

11.5.1 What's Changed

- Move OpenAPI schemas into Python package (#1095)

Changes

11.6 1.3.0 - 27 Apr 2023

> WARNING :warning: : The `1.3.0` has been yanked from PyPi due to a packaging issue. This should have been now resolved in >= `1.3.1`.

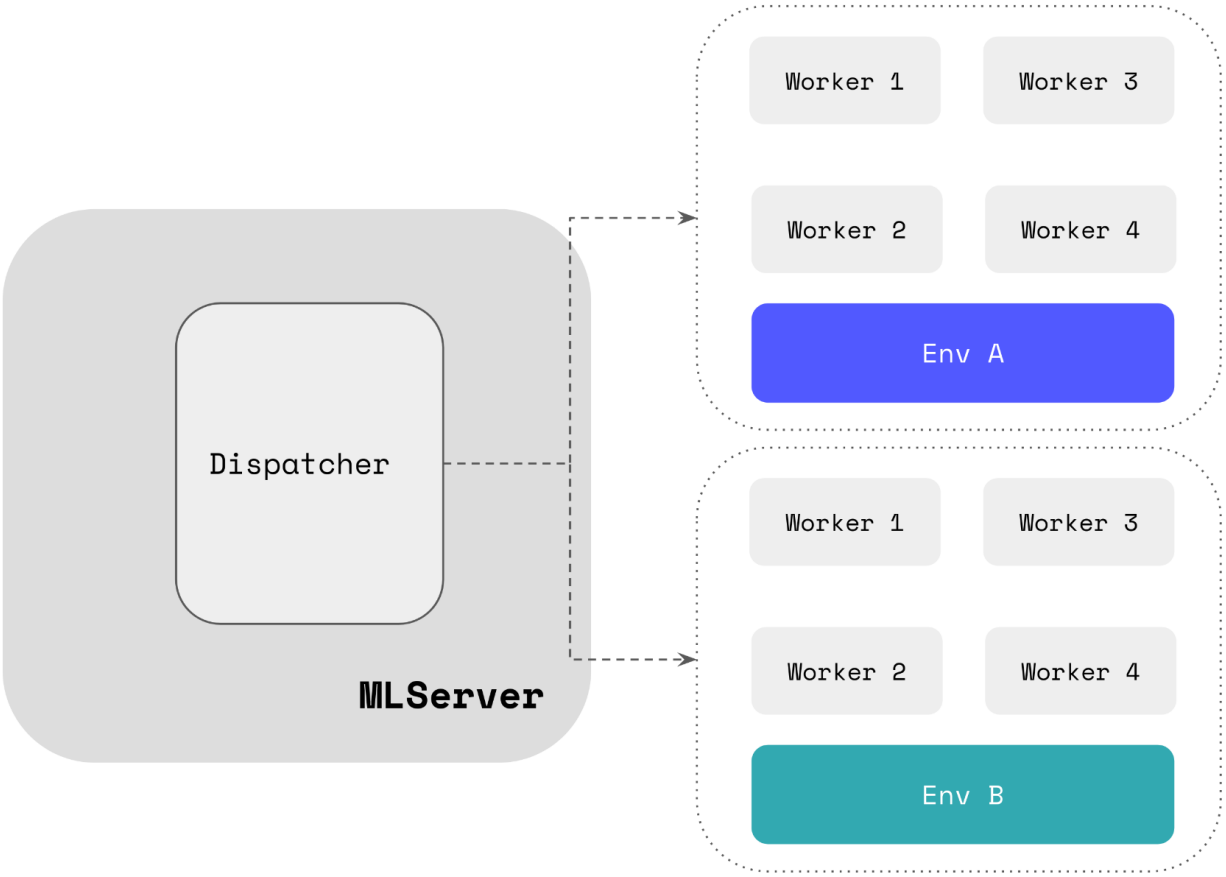
11.6.1 What's Changed

Custom Model Environments

More often that not, your custom runtimes will depend on external 3rd party dependencies which are not included within the main MLServer package - or different versions of the same package (e.g. `scikit-learn==1.1.0` vs `scikit-learn==1.2.0`). In these cases, to load your custom runtime, MLServer will need access to these dependencies.

In MLServer `1.3.0`, it is now possible to load this custom set of dependencies by providing them, through an environment tarball, whose path can be specified within your `model-settings.json` file. This custom environment will get provisioned on the fly after loading a model - alongside the default environment and any other custom environments.

Under the hood, each of these environments will run their own separate pool of workers.



Custom Metrics

The MLServer framework now includes a simple interface that allows you to register and keep track of any custom metrics:

- [mlserver.register()](https://mlserver.readthedocs.io/en/latest/reference/api/metrics.html#mlserver.register): Register a new metric.
- [mlserver.log()](https://mlserver.readthedocs.io/en/latest/reference/api/metrics.html#mlserver.log): Log a new set of metric / value pairs.

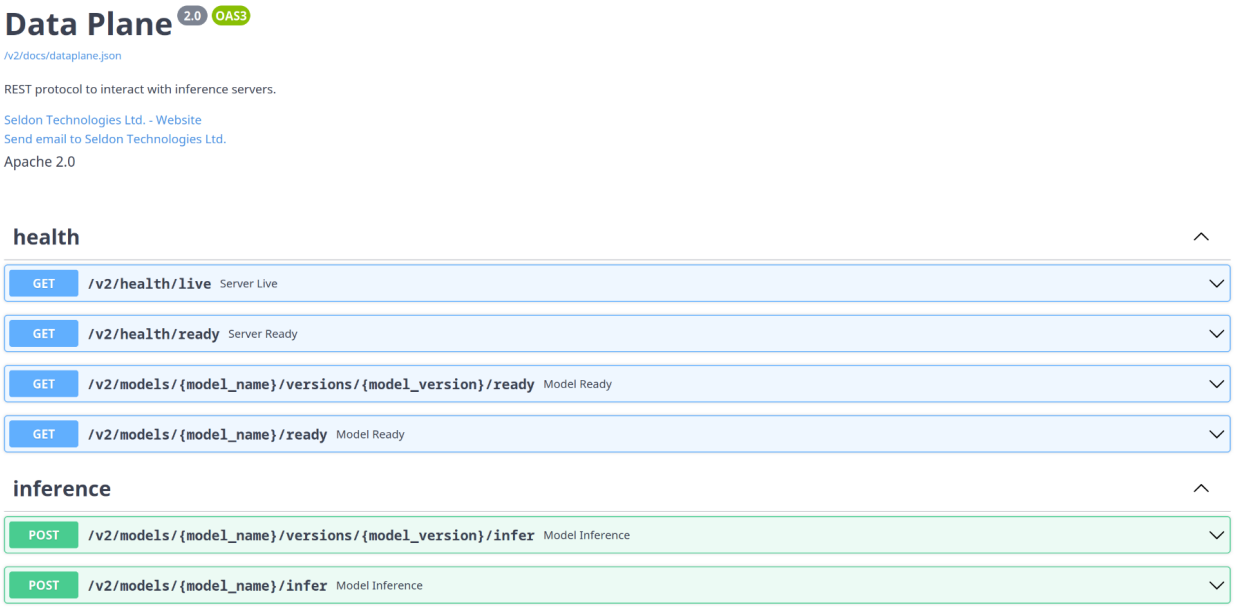
Custom metrics will generally be registered in the [load()](https://mlserver.readthedocs.io/en/latest/reference/api/model.html#mlserver.MLModel.load) method and then used in the [predict()](https://mlserver.readthedocs.io/en/latest/reference/api/model.html#mlserver.MLModel.predict) method of your custom runtime. These metrics can then be polled and queried via Prometheus.

```python
import mlserver

from mlserver.types import InferenceRequest, InferenceResponse

class MyCustomRuntime(mlserver.MLModel):
  async def load(self) -> bool:
    self._model = load_my_custom_model()
    mlserver.register("my_custom_metric", "This is a custom metric example")
    return True

  async def predict(self, payload: InferenceRequest) -> InferenceResponse:
    mlserver.log(my_custom_metric=34)
    # TODO: Replace for custom logic to run inference
    return self._model.predict(payload)
```

OpenAPI

MLServer `1.3.0` now includes an autogenerated Swagger UI which can be used to interact dynamically with the Open Inference Protocol.

The autogenerated Swagger UI can be accessed under the `/v2/docs` endpoint.



Alongside the general API documentation, MLServer also exposes now a set of API docs tailored to individual models, showing the specific endpoints available for each one.

The model-specific autogenerated Swagger UI can be accessed under the following endpoints:

- `/v2/models/{model_name}/docs`
- `/v2/models/{model_name}/versions/{model_version}/docs`

HuggingFace Improvements

MLServer now includes improved Codec support for all the main different types that can be returned by HugginFace models - ensuring that the values returned via the Open Inference Protocol are more semantic and meaningful.

Massive thanks to @pepesi for taking the lead on improving the HuggingFace runtime!

Support for Custom Model Repositories

Internally, MLServer leverages a Model Repository implementation which is used to discover and find different models (and their versions) available to load. The latest version of MLServer will now allow you to swap this for your own model repository implementation - letting you integrate against your own model repository workflows.

This is exposed via the model_repository_implementation flag of your `settings.json` configuration file.

Thanks to @jgallardorama (aka @jgallardorama-itx) for his effort contributing this feature!

Batch and Worker Queue Metrics

MLServer `1.3.0` introduces a new set of metrics to increase visibility around two of its internal queues:

- Adaptive batching queue: used to accumulate request batches on the fly.
- Parallel inference queue: used to send over requests to the inference worker pool.

Many thanks to @alvarorsant for taking the time to implement this highly requested feature!

Image Size Optimisations

The latest version of MLServer includes a few optimisations around image size, which help reduce the size of the official set of images by more than ~60% - making them more convenient to use and integrate within your workloads. In the case of the full `seldonio/mlserver:1.3.0` image (including all runtimes and dependencies), this means going from 10GB down to ~3GB.

Python API Documentation

Alongside its built-in inference runtimes, MLServer also exposes a Python framework that you can use to extend MLServer and write your own codecs and inference runtimes. The MLServer official docs now include a reference page documenting the main components of this framework in more detail.

11.6.2 New Contributors

- @rio made their first contribution in https://github.com/SeldonIO/MLServer/pull/864
- @pepesi made their first contribution in https://github.com/SeldonIO/MLServer/pull/692
- @jgallardorama made their first contribution in https://github.com/SeldonIO/MLServer/pull/849
- @alvarorsant made their first contribution in https://github.com/SeldonIO/MLServer/pull/860
- @gawsoftpl made their first contribution in https://github.com/SeldonIO/MLServer/pull/950
- @stephen37 made their first contribution in https://github.com/SeldonIO/MLServer/pull/1033
- @sauerburger made their first contribution in https://github.com/SeldonIO/MLServer/pull/1064

Changes

11.7 1.2.4 - 10 Mar 2023

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.2.3...1.2.4

Changes

11.8 1.2.3 - 16 Jan 2023

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.2.2...1.2.3

Changes

11.9 1.2.2 - 16 Jan 2023

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.2.1...1.2.2

Changes

11.10 1.2.1 - 19 Dec 2022

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.2.0...1.2.1

Changes

11.11 1.2.0 - 25 Nov 2022

11.11.1 What's Changed

Simplified Interface for Custom Runtimes

MLServer now exposes an alternative *"simplified"* interface which can be used to write custom runtimes. This interface can be enabled by decorating your predict() method with the `mlserver.codecs.decode_args` decorator, and it lets you specify in the method signature both how you want your request payload to be decoded and how to encode the response back.

Based on the information provided in the method signature, MLServer will automatically decode the request payload into the different inputs specified as keyword arguments. Under the hood, this is implemented through MLServer's codecs and content types system.

```python
from mlserver import MLModel
from mlserver.codecs import decode_args

class MyCustomRuntime(MLModel):

  async def load(self) -> bool:
    # TODO: Replace for custom logic to load a model artifact
    self._model = load_my_custom_model()
    self.ready = True
    return self.ready

  @decode_args
  async def predict(self, questions: List[str], context: List[str]) -> np.ndarray:
    # TODO: Replace for custom logic to run inference
    return self._model.predict(questions, context)
```

Built-in Templates for Custom Runtimes

To make it easier to write your own custom runtimes, MLServer now ships with a `mlserver init` command that will generate a templated project. This project will include a skeleton with folders, unit tests, Dockerfiles, etc. for you to fill.

Dynamic Loading of Custom Runtimes

MLServer now lets you load custom runtimes dynamically into a running instance of MLServer. Once you have your custom runtime ready, all you need to do is to move it to your model folder, next to your `model-settings.json` configuration file.

For example, if we assume a flat model repository where each folder represents a model, you would end up with a folder structure like the one below:

```
.
├── models
│   └── sum-model
│       ├── model-settings.json
│       ├── models.py
```

Batch Inference Client

This release of MLServer introduces a new `mlserver infer` command, which will let you run inference over a large batch of input data on the client side. Under the hood, this command will stream a large set of inference requests from specified input file, arrange them in microbatches, orchestrate the request / response lifecycle, and will finally write back the obtained responses into output file.

Parallel Inference Improvements

The `1.2.0` release of MLServer, includes a number of fixes around the parallel inference pool focused on improving the architecture to optimise memory usage and reduce latency. These changes include (but are not limited to):

- The main MLServer process won't load an extra replica of the model anymore. Instead, all computing will occur on the parallel inference pool.
- The worker pool will now ensure that all requests are executed on each worker's AsyncIO loop, thus optimising compute time vs IO time.
- Several improvements around logging from the inference workers.

Dropped support for Python 3.7

MLServer has now dropped support for Python `3.7`. Going forward, only `3.8`, `3.9` and `3.10` will be supported (with `3.8` being used in our official set of images).

Move to UBI Base Images

The official set of MLServer images has now moved to use UBI 9 as a base image. This ensures support to run MLServer in OpenShift clusters, as well as a well-maintained baseline for our images.

Support for MLflow 2.0

In line with MLServer's close relationship with the MLflow team, this release of MLServer introduces support for the recently released MLflow 2.0. This introduces changes to the drop-in MLflow "scoring protocol" support, in the MLflow runtime for MLServer, to ensure it's aligned with MLflow 2.0.

MLServer is also shipped as a dependency of MLflow, therefore you can try it out today by installing MLflow as:

```
$ pip install mlflow[extras]
```

To learn more about how to use MLServer directly from the MLflow CLI, check out the MLflow docs.

11.11.2 New Contributors

- @johnpaulett made their first contribution in https://github.com/SeldonIO/MLServer/pull/633
- @saeid93 made their first contribution in https://github.com/SeldonIO/MLServer/pull/711
- @RafalSkolasinski made their first contribution in https://github.com/SeldonIO/MLServer/pull/720
- @dumaas made their first contribution in https://github.com/SeldonIO/MLServer/pull/742
- @Salehbigdeli made their first contribution in https://github.com/SeldonIO/MLServer/pull/776
- @regen100 made their first contribution in https://github.com/SeldonIO/MLServer/pull/839

Full Changelog: https://github.com/SeldonIO/MLServer/compare/1.1.0...1.2.0

Changes

11.12 v1.2.0.dev1 - 01 Aug 2022

Changes

11.13 v1.1.0 - 01 Aug 2022

Changes

CHAPTER

TWELVE

MLSERVER

An open source inference server for your machine learning models.

12.1 Overview

MLServer aims to provide an easy way to start serving your machine learning models through a REST and gRPC interface, fully compliant with KFServing's V2 Dataplane spec. Watch a quick video introducing the project here.

- Multi-model serving, letting users run multiple models within the same process.
- Ability to run inference in parallel for vertical scaling across multiple models through a pool of inference workers.
- Support for adaptive batching, to group inference requests together on the fly.
- Scalability with deployment in Kubernetes native frameworks, including Seldon Core and KServe (formerly known as KFServing), where MLServer is the core Python inference server used to serve machine learning models.
- Support for the standard V2 Inference Protocol on both the gRPC and REST flavours, which has been standardised and adopted by various model serving frameworks.

You can read more about the goals of this project on the inital design document.

12.2 Usage

You can install the `mlserver` package running:

```
pip install mlserver
```

Note that to use any of the optional *inference runtimes*, you'll need to install the relevant package. For example, to serve a `scikit-learn` model, you would need to install the `mlserver-sklearn` package:

```
pip install mlserver-sklearn
```

For further information on how to use MLServer, you can check any of the *available examples*.

12.3 Inference Runtimes

Inference runtimes allow you to define how your model should be used within MLServer. You can think of them as the **backend glue** between MLServer and your machine learning framework of choice. You can read more about *inference runtimes in their documentation page*.

Out of the box, MLServer comes with a set of pre-packaged runtimes which let you interact with a subset of common frameworks. This allows you to start serving models saved in these frameworks straight away. However, it's also possible to ***write custom runtimes***.

Out of the box, MLServer provides support for:

Framework	Supported	Documentation
Scikit-Learn		*MLServer SKLearn*
XGBoost		*MLServer XGBoost*
Spark MLlib		*MLServer MLlib*
LightGBM		*MLServer LightGBM*
Tempo		`github.com/SeldonIO/tempo`
MLflow		*MLServer MLflow*
Alibi-Detect		*MLServer Alibi Detect*
Alibi-Explain		*MLServer Alibi Explain*
HuggingFace		*MLServer HuggingFace*

12.4 Examples

To see MLServer in action, check out *our full list of examples*. You can find below a few selected examples showcasing how you can leverage MLServer to start serving your machine learning models.

- *Serving a `scikit-learn` model*
- *Serving a `xgboost` model*
- *Serving a `lightgbm` model*
- *Serving a `tempo` pipeline*
- *Serving a custom model*
- *Serving an `alibi-detect` model*

- *Serving a `HuggingFace` model*
- *Multi-Model Serving with multiple frameworks*
- *Loading / unloading models from a model repository*

12.5 Developer Guide

12.5.1 Versioning

Both the main `mlserver` package and the *inference runtimes packages* try to follow the same versioning schema. To bump the version across all of them, you can use the *`./hack/update-version.sh`* script.

For example:

```
./hack/update-version.sh 0.2.0.dev1
```

BIBLIOGRAPHY

[ZWCS20] Jiale Zhi, Rui Wang, Jeff Clune, and Kenneth O. Stanley. Fiber: A Platform for Efficient Development and Distributed Training for Reinforcement Learning and Population-Based Methods. *arXiv:2003.11164 [cs, stat]*, March 2020. arXiv:2003.11164.

PYTHON MODULE INDEX

m

INDEX

Symbols

B

C

D

E

N

O

P

R

S

T

U

V

W