MLServer

Release 1.1.0.dev6

Seldon Technologies

Jul 05, 2022

CONTENTS

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

CHAPTER

ONE

CONTENT TYPES

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(payload: Any) -> InferenceRequest`
  - `decode_request(request: InferenceRequest) -> Any`

  - `encode_response(model_name: str, payload: Any, model_version: str) -> InferenceResponse`
  - `decode_response(response: InferenceResponse) -> Any`
- **Input Codecs**
  - `encode_input(name: str, payload: Any) -> RequestInput`
  - `decode_input(request_input: RequestInput) -> Any`
  - `encode_output(name: str, payload: Any) -> ResponseOutput`
  - `decode_output(response_output: ResponseOutput) -> Any`

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]})

v2_request = PandasCodec.encode_request(dataframe)
print(v2_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*.

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],
    },
    {
```

(continues on next page)

(continued from previous page)

```
      "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.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.

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
v2_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
v2_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"]
})

v2_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 import StringRequestCodec

# Encode an entire V2 request
v2_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
v2_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
v2_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
v2_request = InferenceRequest(
  inputs=[
    DatetimeCodec.encode_input("foo", foo, use_bytes=False)
  ]
)
```

CHAPTER

TWO

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.

2.1 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.

2.1.1 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.

2.2 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.

2.2.1 `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.

2.3 References

CHAPTER

THREE

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".

3.1 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**.

3.2 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.

3.2.1 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.

3.2.2 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.

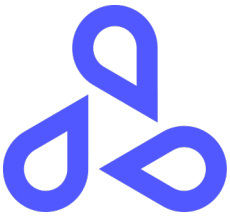
CHAPTER

FOUR

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

4.1 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.

4.1.1 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: kfserving
  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.

4.1.2 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: kfserving
  predictors:
    - name: default
      graph:
        name: classifier
      componentSpecs:
        - spec:
            containers:
              - name: classifier
                image: my-custom-server:0.1.0
                ports:
                  - containerPort: 8080
                    name: http
                    protocol: TCP
                  - containerPort: 8081
```

(continues on next page)

(continued from previous page)

```
                    name: grpc
                    protocol: TCP
```

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 `kserving`.
- Pointing our model container to use our **custom MLServer image**, by specifying it on the `image` field of the `componentSpecs` section of the manifest.
- Adding a couple workarounds to ensure that the serving environment used by Seldon Core is compatible with the custom MLServer images.

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
```

4.2 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.

4.2.1 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.

4.2.2 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:
```

(continues on next page)

(continued from previous page)

```
        - 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

FIVE

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.

5.1 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-huggingface`	`mlserver_huggingface.HuggingFaceRuntime`	HuggingFace example	*MLServer HuggingFace*
Spark MLlib	`mlserver-mllib`	`mlserver_mllib.MLlibModel`	Coming Soon	*MLServer MLlib*
LightGBM	`mlserver-lightgbm`	`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-detect`	`mlserver_alibi_detect.AlibiDetectRuntime`	*Alibi-detect example*	*MLServer Alibi-Detect*

5.1.1 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*.

5.1.2 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*.

Content Types

The XGBoost runtime supports a new `dmatrix` content type, aim to decode V2 Inference payloads into XGBoost's `DMatrix` data type. This content type uses a similar set of encoding rules as the *NumPy Array one*.

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

5.1.3 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.

5.1.4 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 github.com/SeldonIO/MLServer repository.

5.1.5 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*.

5.1.6 Alibi-Detect runtime for MLServer

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

Usage

You can install the 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*.

5.1.7 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.

5.1.8 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.

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 `mlserver.MLModel` abstract class, whose main methods are:

- `load(self) -> bool`: Responsible for loading any artifacts related to a model (e.g. model weights, pickle files, etc.).
- `predict(self, payload: InferenceRequest) -> InferenceResponse`: 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 `mlserver.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()
    self.ready = True
    return self.ready

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

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"})
    )
```

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.

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.

Additionally, the `mlserver build` subcommand will also 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.

Docker-less Environments

In some occasions, it may not be possible to access an environment with a running Docker daemon. This can be the case, for example, on some CI pipelines.

To account for these use cases, MLServer also includes a `mlserver dockerfile` subcommand which will just generate a `Dockerfile` (and optionally a `.dockerignore` file). This `Dockerfile` can be then by used by other *"Docker-less"* tools, like Kaniko or Buildah to build the final image.

CHAPTER

SIX

REFERENCE

6.1 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`).

6.1.1 Settings

pydantic settings mlserver.settings.**Settings**

Config

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

Fields

- *cors_settings (Optional[mlserver.settings.CORSSettings])*
- *debug (bool)*
- *extensions (List[str])*
- *grpc_max_message_length (Optional[int])*
- *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 (Optional[str])*
- *metrics_endpoint (Optional[str])*
- *metrics_port (int)*

- *model_repository_root (str)*
- *parallel_workers (int)*
- *root_path (str)*
- *server_name (str)*
- *server_version (str)*

field cors_settings: Optional[mlserver.settings.CORSSettings] = None

field debug: bool = True

field extensions: List[str] = []
Server extensions loaded.

field grpc_max_message_length: Optional[int] = 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: Optional[str] = None
Path to logging config file.

field metrics_endpoint: Optional[str] = '/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 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 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 = '1.1.0.dev6'
Version of the server.

6.2 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.

6.2.1 Settings

pydantic settings mlserver.settings.**ModelSettings**

Config

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

Fields

- *implementation (pydantic.types.PyObject)*
- *inputs (List[mlserver.types.dataplane.MetadataTensor])*
- *max_batch_size (int)*
- *max_batch_time (float)*
- *name (str)*
- *outputs (List[mlserver.types.dataplane.MetadataTensor])*
- *parallel_workers (Optional[int])*
- *parameters (Optional[mlserver.settings.ModelParameters])*
- *platform (str)*
- *versions (List[str])*
- *warm_workers (bool)*

field implementation: pydantic.types.PyObject = 'mlserver.model.MLModel'

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

field inputs: List[mlserver.types.dataplane.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[mlserver.types.dataplane.MetadataTensor] = []

Metadata about the outputs returned by the model.

field parallel_workers: Optional[int] = None

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

field parameters: Optional[*mlserver.settings.ModelParameters*] = 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.

6.2.2 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_*

Fields

- *content_type (Optional[str])*
- *extra (Optional[dict])*
- *format (Optional[str])*
- *uri (Optional[str])*
- *version (Optional[str])*

field content_type: Optional[str] = None

Default content type to use for requests and responses.

field extra: Optional[dict] = {}

Arbitrary settings, dependent on the inference runtime implementation.

field format: Optional[str] = None

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

field uri: Optional[str] = None

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

field version: Optional[str] = None

Version of the model.

6.3 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
```

6.3.1 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

start

Start serving a machine learning model with MLServer.

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

Arguments

FOLDER

Required argument

CHAPTER

SEVEN

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.

7.1 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*

7.1.1 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
from sklearn import datasets, svm, metrics
```

(continues on next page)

(continued from previous page)

```
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()
        }
    ]
}

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]
```

7.1.2 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)

    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)

# read in data
```

(continues on next page)

(continued from previous page)

```
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.

NOTE

You may first need to install the XGBoost inference runtime for MLServer using `pip install mlserver-xgboost`

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.toarray().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]
```

```

```

7.1.3 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:8788/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]
```

7.1.4 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()
```

```

```

7.1.5 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.

```
[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",
          "data": [0.9978],
        },
        {
```

(continues on next page)

(continued from previous page)

```
          "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 = {
    "columns": [
        "alcohol",
        "chlorides",
        "citric acid",
        "density",
        "fixed acidity",
        "free sulfur dioxide",
        "pH",
        "residual sugar",
        "sulphates",
        "total sulfur dioxide",
        "volatile acidity",
    ],
```

(continues on next page)

(continued from previous page)

```
    "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.

```

```

7.1.6 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., 0.2))
    M, A = 0., 0.
    if marriage is not None:
        bM = numpyro.sample('bM', dist.Normal(0., 0.5))
        M = bM * marriage
    if age is not None:
        bA = numpyro.sample('bA', dist.Normal(0., 0.5))
        A = bA * age
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    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)
```

(continues on next page)

(continued from previous page)

```
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.

```
%%writefile models.py
import json
import numpyro
import numpy as np

from typing import Dict
from jax import random
```

(continues on next page)

(continued from previous page)

```
from mlserver import MLModel, types
from mlserver.utils import get_model_uri
from numpyro.infer import Predictive
from numpyro import distributions as dist


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)

        self.ready = True
        return self.ready

    async def predict(self, payload: types.InferenceRequest) -> types.InferenceResponse:
        inputs = self._extract_inputs(payload)
        predictions = self._predictive(rng_key=random.PRNGKey(0), **inputs)

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

        return types.InferenceResponse(
            id=payload.id,
            model_name=self.name,
            model_version=self.version,
            outputs=[
                types.ResponseOutput(
                    name="obs_mean",
                    shape=obs_mean.shape,
                    datatype="FP32",
                    data=np.asarray(obs_mean).tolist(),
                )
            ],
        )

    def _extract_inputs(self, payload: types.InferenceRequest) -> Dict[str, np.ndarray]:
        inputs = {}
        for inp in payload.inputs:
            inputs[inp.name] = np.array(inp.data)

        return inputs

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

(continues on next page)

(continued from previous page)

```
        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

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

model-settings.json

```
%%writefile 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

x_0 = [28.0]
inference_request = {
    "inputs": [
        {
          "name": "marriage",
          "shape": [1],
          "datatype": "FP32",
          "data": x_0
        }
    ]
}

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

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.

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 requests

x_0 = [28.0]
inference_request = {
    "inputs": [
        {
          "name": "marriage",
          "shape": [1],
          "datatype": "FP32",
          "data": x_0
        }
    ]
}

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

print(response)
print(response.text)
```

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.

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:
```

(continues on next page)

(continued from previous page)

```
            - name: numpyro-divorce
              image: my-custom-numpyro-server:0.1.0
```

```

```

7.1.7 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
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
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": [
```

(continues on next page)

(continued from previous page)

```
        {
            "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
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
```

7.2 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*

7.2.1 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-learn`	MNIST example from the `scikit-learn` documentation	`./models/mnist-svm/model.joblib`
`mushroom-xgboost`	`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))
```

(continues on next page)

(continued from previous page)

```
# 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/
↪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
```

(continues on next page)

(continued from previous page)

```
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)

# 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",
          "data": x_0.toarray().tolist()
        }
    ]
}

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

response.json()
```

```

```

7.2.2 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()
```

```

```

7.2.3 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"}}}
}

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)
```

(continues on next page)

(continued from previous page)

```
            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"
            }
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "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
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(
```

(continues on next page)

(continued from previous page)

```
    "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": {
                "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",
```

(continues on next page)

(continued from previous page)

```
            "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
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}}},
}
```

(continues on next page)

(continued from previous page)

```python
@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)

    @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)
```

(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)
```

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,
                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],
```

(continues on next page)

(continued from previous page)

```
            "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

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) ------")
```

(continues on next page)

(continued from previous page)

```
        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",
            "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",
```

(continues on next page)

(continued from previous page)

```
    json=payload
)
```

```

```

7.2.4 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
dependencies:
  - python == 3.7
  - scikit-learn == 0.20.3
  - joblib == 0.13.0
  - requests
  - pip:
      - mlserver == 0.6.0.dev0
      - mlserver-sklearn ==0.6.0.dev0
```

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:0.6.0.dev0
```

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 = {
    "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/infer"
response = requests.post(endpoint, json=inference_request)

response.json()
```

7.2.5 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


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:
```

(continues on next page)

(continued from previous page)

```
        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

inputs = {
    "name": "Foo Bar",
    "message": "Hello from Client (REST)!"
}

# 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)
```

(continues on next page)

(continued from previous page)

```
response.json()
```

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
import mlserver.grpc.converters as converters
import mlserver.grpc.dataplane_pb2_grpc as dataplane
import mlserver.types as types

model_name = "json-hello-world"
inputs = {
    "name": "Foo Bar",
    "message": "Hello from Client (gRPC)!"
}
inputs_bytes = json.dumps(inputs).encode("UTF-8")

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)
response
```

7.2.6 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 0Xn8RTSlQdmxwjhKnSB_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
```

(continues on next page)

(continued from previous page)

```
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",
}

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.

```
```

CHAPTER

EIGHT

MLSERVER

An open source inference server for your machine learning models.

8.1 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.

8.2 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.

8.3 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*

8.4 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*
- *Multi-Model Serving with multiple frameworks*
- *Loading / unloading models from a model repository*

8.5 Developer Guide

8.5.1 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.

INDEX

Symbols

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

U

V

W