mlserving Documentation

Release 0.1.0

Or Levi

Aug 28, 2020

Getting Started

Stable Version

Source Code https://github.com/orlevii/mlserving

mlserving is a framework for developing a realtime model-inference service.

Allows you to easily set-up an inference-endpoint for your ML Model.

mlserving emphasizes on high performance and allows easy integration with other model servers such as **TensorFlow Serving**

CHAPTER 1

Motivation

Data Scientists usually struggle with integrating their ML-models to production.

mlserving is here to make the development of model-servers easy for everyone.

CHAPTER 2

Demo Repository

SKLearn model: https://github.com/orlevii/mlserving-example

2.1 Installation

mlserving requires python >= 3.6

Installation is simple:

```
$ pip install mlserving
```

2.2 Serving Models

Serving models is easy, by implementing simple interfaces, you can set up endpoints in no time. Just implement your business-logic, mlserving takes care of everything else.

Simple example of serving scikit-learn `LogisticRegression` model

```
from mlserving import ServingApp
from mlserving.predictors import RESTPredictor

import joblib # for deserialization saved models


class MyPredictor(RESTPredictor):
    def __init__(self):
        # Loading a saved model
        self.model = joblib.load('./models/logistic_regression.pkl')

    def pre_process(self, input_data, req):
        return input_data['features']
```

(continues on next page)

(continued from previous page)

```
    def predict(self, processed_data, req):
        return self.model.predict_proba(processed_data)[0]

    def post_process(self, prediction, req):
        return {'probability': prediction}

app = ServingApp()
app.add_inference_handler('/api/v1/predict', MyPredictor())
app.run()
```

This example assumes your endpoint receives post-processed features.

app.run() - Will start up development server, by default it listens on port 5000

2.3 Predictors

PredictorBase presents a simple interface:

- **before_request** - Things to do before starting the ml-business logics (like parsing, validations, etc...)
- **pre_process** - Used for feature processing (string embedding, normalizations, etc...)
- **predict** - Predicting a result with your loaded model
- **post_process** - Formatting a response for your client, the return value of this method will be sent back in the response

This interface is very intuitive, and makes it easy to integrate other serving-apps like `TensorFlow Serving`.

These methods are called one after the other, the output of a method will become the input of the next one in line.

Since the most common format of transmitting data over HTTP/1.1 is JSON, mlserving accepts & returns JSONs only

2.3.1 RESTPredictor

This class implements PredictorBase and the go-to class to inherit from.

When you inherit from RESTPredictor, you usually want to load relevant resources on `def __init__`

```
from mlserving.predictors import RESTPredictor

import joblib

class MyPredictor(RESTPredictor):
    def __init__(self):
        # Loading resources
        self.model = joblib.load('./models/my_trained_model.pkl')
```

RESTPredictor also adds validations to the input request in `before_request`

The validation is done with ***validr***, a fast (and easy to use python library).

In order to define your request schema, you'll need to add a decorator above your predictor class:

```
from mlserving.api import request_schema
from mlserving.predictors import RESTPredictor

SCHEMA = {
    # floats only list
    'features': [
        'list',
        'float'
    ]
}

@request_schema(SCHEMA)
class MyPredictor(RESTPredictor):
    # TODO: Implement "def predict" & override methods (if needed)
    pass
```

validr syntax can be found here: https://github.com/guyskk/validr/wiki/Schema-Syntax

2.3.2 PipelinePredictor

Whenever your prediction is based on the result of several models, you should consider using PipelinePredictor for chaining models one after the other.

A good example would be a text classification model.

Request with input text -> text processing -> embedding -> classification

```
from mlserving import ServingApp
from mlserving.api import request_schema
from mlserving.predictors import RESTPredictor, PipelinePredictor

SCHEMA = {
    'text': 'str'
}

@request_schema(SCHEMA)
class EmbeddingPredictor(RESTPredictor):
    def __init__(self):
        # Load relevant resources
        pass

    def pre_process(self, features: dict, req):
        text = features['text']
        # Clean the text, make other processing if needed
        return text

    def predict(self, processed_text, req):
        # Use the processed_text and get its embedding
        pass

class TextClassifierPredictor(RESTPredictor):
    def __init__(self):
        # Load relevant resources
        pass

     def predict(self, features: dict, req):
        # Make the prediction based on the text-embedding
```

(continues on next page)

(continued from previous page)

```
        pass


app = ServingApp()
p = PipelinePredictor([
        EmbeddingPredictor(),
        TextClassifierPredictor()
    ])

app.add_inference_handler(p, '/classify_text')
```

2.4 TensorFlow Serving

2.4.1 Intro

TensorFlow Serving is an high-performance system designed for serving TensorFlow models.

It can load saved models (ProtoBuff format) and expose an endpoint for inference

Sometimes, we can't use TensorFlow serving alone as we need to make some processing before/after the inference.

Read more about TensorFlow Serving: https://www.tensorflow.org/tfx/guide/serving

2.4.2 Integration

mlserving allows easy integration with TensorFlow Serving model server.

The idea is to have a python layer that makes some processing before invoking the tf-serving endpoint.

TFServingPrediction implements def predict can be used as a mixin that handles the tf-serving request

`requests` package is required for TFServingPrediction to work properly

```
$ pip install requests
```

```
from mlserving.predictors import RESTPredictor
from mlserving.predictors.tensorflow import TFServingPrediction

class MyPredictor(TFServingPrediction, RESTPredictor):
    def __init__(self):
        # configure the TFServingPrediction with default values.
        super().__init__()
        # Default values: host='127.0.0.1' port=8501 model_name='model'

    def pre_process(self, features: dict, req):
        return {
            "instances": [
                # TODO: fill your tensor inputs here
            ]
        }

    def post_process(self, prediction, req):
        prediction = prediction['prediction']
        return {
```

(continues on next page)

(continued from previous page)

```
            'probabilities': prediction,
        }
```

Since def predict is already implemented, we just need to implement the processing layer that comes before/after the inference

2.5 Health Check

ServingApp allows to add additional `GET` route for handling health/ping requests

```
from mlserving import ServingApp

app = ServingApp()

# Using the default health handler
app.add_health_handler('/ping')

app.run()
```

The default handler always returns `200 OK`

ServingApp listens to SIGTERM signal, if SIGTERM signal was sent to the process, all the health routes will start returning `503` (Useful when making a graceful shutdown to the application)

2.5.1 Custom Health Handlers

app.add_health_handler 2nd argument is an handler, so you just need to implement one.

```
from mlserving import ServingApp
from mlserving.health import HealthHandler, Healthy, Unhealthy

class CustomHealthHandler(HealthHandler):
        def health_check(self):
            if some_condition:
                return Unhealthy()
            return Healthy()

app = ServingApp()

# Using our custom health handler
app.add_health_handler('/ping', CustomHealthHandler())

app.run()
```

2.6 Running In Production

Although you can use app.run() in order start up your service, it is only recommended for local development.

We encourage to use gunicron for production use.

gunicorn with gevent is battle-tested works well for most use-cases:

2.6.1 Gunicorn Installation

```
$ pip install gunicorn[gevent]
```

2.6.2 Example

Serve your application: `app.py`:

```
# app.py example

from mlserving import ServingApp
# other imports ...


app = ServingApp()
```

Run: $ gunicorn -b 0.0.0.0:5000 -k gevent -w 4 app:app

Read more here

2.7 Web Frameworks

Currently, ***falcon*** is the only WebFramework implemented.

You can implement your own web-framework (if you need to) and pass it as a parameter.

```
from mlserving import ServingApp
from mlserving.webframeworks import WebFramework

class MyWebFramework(WebFramework):
    #TODO: Implement abstract methods...
    pass

app = ServingApp(web_framework=MyWebFramework())
```