BentoML

Aug 16, 2019

Contents

BentoML is a python framework for building, shipping and running machine learning services. It provides high-level APIs for defining an ML service and packaging its artifacts, source code, dependencies, and configurations into a production-system-friendly format that is ready for deployment.

Use BentoML if you need to:

- Turn your ML model into REST API server, Serverless endpoint, PyPI package, or CLI tool
- Manage the workflow of creating and deploying a ML service

CHAPTER 1

Getting Started

Defining a machine learning service with BentoML is as simple as a few lines of code:

```
@artifacts([PickleArtifact('model')])
@env(conda_pip_dependencies=["scikit-learn"])
class IrisClassifier(BentoService):

    @api(DataframeHandler)
    def predict(self, df):
        return self.artifacts.model.predict(df)
```

CHAPTER 2

Feature Highlights

- **Multiple Distribution Formats** - Easily package machine learning models and preprocessing code into a format that works best with your inference scenarios:
  - Docker Image - Deploy as container running REST API server
  - PyPi Package - Integrate into python applications seamlessly
  - CLI tool - Incorporate model into Airflow DAG or CI/CD pipeline
  - Spark UDF - Run batch inference on a large dataset with Spark
  - Serverless Function - Host model on serverless platforms such as AWS Lambda
- **Multiple Frameworks Support** - BentoML supports a wild range of machine learning frameworks out-of-box including Tensorflow, PyTorch, Scikit-Learn, xgboost, H2O, FastAI and can be easily extended to work with new or custom frameworks.
- **Deploy Anywhere** - BentoML bundled machine learning service can be easily deployed with platforms such as Docker, Kubernetes, Serverless, Airflow and Clipper, on cloud providers including AWS, Google Cloud, and Azure.
- **Custom Runtime Backend** - Easily integrate python preprocessing code with high performance deep learning runtime backend such as Tensorflow-serving.

CHAPTER 3

Content

3.1 Quick Start

3.1.1 Install BentoML

Install BentoML is straightforward.

```
pip install bentoml
```

3.1.2 Running the quick start project

The easiest way to try out the quick start project is using Google's Colab to run the quick start project.

You can also run the quick start project locally. Download the quick start example by git clone the BentoML repo, and navigate to the quick start project inside the *example* folder

```
$ pip install jupyter
$ git clone http://github.com/bentoml/bentoml
$ cd bentoml
$ jupyter notebook examples/quick-start/bentoml-quick-start-guide.ipynb
```

We will go through each cell inside the notebook with explanation follows below.

3.1.3 Quick start walk through

3.1.3.1 Add BentoML to the notebook and training classification model

```
!pip install -I bentoml
!pip install pandas sklearn
```

We use jupyter notebook's built-in magic command to download and install python modules such as scikit-learn for our example model. We also download and install BentoML for define ML service later on.

```
from sklearn import svm
from sklearn import datasets

clf = svm.SVC(gamma='scale')
iris = datasets.load_iris()
X, y = iris.data, iris.target
clf.fit(X, y)
```

We trained a classification model with scikit-learn's iris dataset.

3.1.3.2 Define machine learning service with BentoML

```
1  %%writefile iris_classifier.py
2  from bentoml import BentoService, api, env, artifacts
3  from bentoml.artifact import PickleArtifact
4  from bentoml.handlers import DataframeHandler
5
6  @artifacts([PickleArtifact('model')])
7  @env(conda_pip_dependencies=["scikit-learn"])
8  class IrisClassifier(BentoService):
9
10     @api(DataframeHandler)
11     def predict(self, df):
12         return self.artifacts.model.predict(df)
```

Line 1: We use jupyter notebook's built-in magic command to save our ML service definition into a python file

Line 2: We import BentoService, our ML service will build on top of this by subclassing it. We also import decorators such as, artifacts, api and env for defining our ML service.

- **artifacts** decorator define what artifacts are required for packaging this service.
- **env** decorator designed for specifying the desired system environment and dependencies in order for this service to load. For this project we are using conda environment. If you already have requirement.txt file listing all of the python libraries you need:

  ```
  @env(requirement_txt="../my_project/requirement.txt")
  ```

- **api**: decorator allow us to add an entry point to accessing this service. Each *api* will be translated into a REST endpoint when deploying as API server, or a CLI command when running the service as CLI tool.

Line 3: Using PickleArtifact for packaging our classifier model. Beside PickleArtifact, BentoML offers *KerasModelArtifact*, *PytorchModelArtifact*, *H2oModelArtifact*, *XgboostModelArtifact* and etc.

Line 4: Each API endpoint requires a Handler for defining the expect input format. For this project, we are using **DataframeHandler** to transform either a HTTP request or CLI command argument into a pandas dataframe and pass it down to the user defined API function. BentoML also provides *JsonHandler*, *ImageHandler* and *TensorHandler*

Line 6-7: We defined what artifact need to be included for this service, and giving it a name *model*, and include the python library that we need for this project.

Line 8: We created our ML service called IrisClassifier by subclassing *BentoService*

Line 10-12: We defined a function called *predict*. It will return result from the artifact, *model*, we defined earlier by calling *predict* on that artifact. We expose this predict function as our api for the service with the *api* decorator, and

tell BentoML that the incoming data will be transformed into pandas dataframe for the user defined *predict* function to consume.

Now we have defined the ML service with BentoML, we will package our trained model next and save it as archive to the file system.

3.1.3.3 Save defined ML service as BentoML service archive

```
1 from iris_classifier import IrisClassifier
2
3 svc = IrisClassifier.pack(model=clf)
4 saved_path = svc.save('/tmp/bentoml_archive')
```

Line 1: We import the service definition we wrote in the previous cell.

Line 3: We are packaging the trained model from above with the ML service.

Line 4-5: We saved the packed service as BentoML archive into the local file system and print out the saved location path.

We just created and saved our quick start project into BentoML service archive. It is a directory containing all of the source code, data, and configurations that required to load and run as Bento Service. You will find three *magic* files that generated within the archive directory:

- bentoml.yml: A YAML file contains all of the metadata related to this service and archive.
- setup.py: The configuration file that makes this BentoML service archive 'pip' installable
- Dockerfile: for building Docker image that expose this Bento service as REST API service.

3.1.3.4 Using BentoML archive

Real-time serving with REST API

To exposing your ML service as HTTP API endpoint, you can simply use the bentoml serve command:

```
!bentoml serve {saved_path}
```

With *bentoml serve* command, a web server will start locally at the port 5000. We created additional endpoints that make this server ready for production.

- */*: The index page with OpenAPI definition.
- */docs.json*: The Open API definition for all endpoints in JSON format.
- */metrics*: Expose system and latency metrics with Prometheus.
- */healthz*: Check on your service health.
- */feedback*: Add business feedback for the predicted results.

Open http://127.0.0.1:5000 to view the documentation for all API endpoints.

Run REST API server with Docker

To deploy the Bento service as REST api server for production use, we can use the generated Dockerfile to create Docker image for that.

```
!cd {saved_path} && docker build -t iris-classifier .
```

```
!docker run -p 5000:5000 iris-classifier
```

Note: To generate Docker image, you will need to install Docker on your system. Please follow direction from this link: https://docs.docker.com/install

(Optional) Get a Client SDK for the above REST API server

To get a client SDK, copy the content of http://127.0.0.1:5000/docs.json and paste to https://editor.swagger.io then click the tab Generate Client and choose the language.

Currently, https://editor.swagger.io supports to generate a client SDK in Java, Kotlin, Swift, Python, PHP, Scala... ect.

Loading Bento service archive in Python

The easiest to use Bento service archive in your python application is using *bentoml.load*.

```
import bentoml
import pandas as pd

bento_svc = bentoml.load(saved_path)
bento_svc.predict([X[0]]
```

pip install a BentoML service archive

BentoML support distributing Bento service as PyPi package, with the generated *setup.py* file. Bento service archive can be installed with pip:

```
!pip install {saved_path}
```

Bento service archive can be uploaded to pypi.org as public python package or to your organization's private PyPi index for all developers in your org to use.

```
!cd {saved_path} & python setup.py sdist upload
```

Note: You will have to configure ".pypirc" file before uploading to pypi index. You can find more information about distributing python package at: https://docs.python.org/3.7/distributing/index.html#distributing-index

After pip install, we can import the Bento service as regular python package.

```
import IrisClassifier

installed_svc = IrisClassifier.load()
installed_svc.predict([X[0]]
```

CLI access with BentoML service archive

pip install includes a CLI tool for accessing the Bento service.

From terminal, you can use *info* command to list all APIs defined in the service.

```
!IrisClassifier info
```

You can use *docs* command to get all APIs in OpenAPI format.

```
!IrisClassifier docs
```

Call prediction with user defined API function.

```
!IrisClassifier predict --help
```

```
!IrisClassifier predict --input='[[5.1, 3.5, 1.4, 0.2]]'
```

Alternatively, use `bentoml cli` to load and run Bento service archive without installing.

```
!bentoml info {saved_path}
```

```
!bentoml docs {saved_path}
```

```
!bentoml predict {saved_path} --input='[[5.1, 3.5, 1.4, 0.2]]'
```

Congratulation! You've train, build, and running your first Bento service.

3.2 API Reference

3.2.1 BentoML

3.2.1.1 BentoService

class bentoml.service.**BentoService**(*artifacts=None*, *env=None*)

BentoService packs a list of artifacts and exposes service APIs for BentoAPIServer and BentoCLI to execute. By subclassing BentoService, users can customize the artifacts and environments required for a ML service.

```
>>>  from bentoml import BentoService, env, api, artifacts, ver
>>>  from bentoml.handlers import DataframeHandler
>>>
>>>  @ver(major=1, minor=4)
>>>  @artifacts([PickleArtifact('clf')])
>>>  @env(conda_dependencies: [ 'scikit-learn' ])
>>>  class MyMLService(BentoService):
>>>
>>>     @api(DataframeHandler)
>>>     def predict(self, df):
>>>         return self.artifacts.clf.predict(df)
>>>
>>>  bento_service = MyMLService.pack(clf=my_trained_clf_object)
>>>  bentoml.save(bento_service, './export')
```

classmethod name()
return bento service name

version()
return bento service version str

get_service_apis()
Return a list of user defined API functions

Returns List of user defined API functions

Return type list(*BentoServiceAPI*)

class bentoml.service.**BentoServiceAPI**(*service*, *name*, *doc*, *handler*, *func*)
BentoServiceAPI defines abstraction for an API call that can be executed with BentoAPIServer and BentoCLI

Parameters

- **service** (BentoService) – ref to service containing this API
- **name** (*str*) – API name, by default this is the python function name
- **handler** (*bentoml.handlers.BentoHandler*) – A BentoHandler class that transforms HTTP Request and/or CLI options into expected format for the API func
- **func** (*function*) – API func contains the actual API callback, this is typically the 'predict' method on a model

3.2.1.2 api

bentoml.**api**(*handler_cls*, **args*, ***kwargs*)
Decorator for adding api to a BentoService

Parameters

- **handler_cls** (*bentoml.handlers.BentoHandler*) – The handler class for the API function.
- **api_name** (str, optional) – API name to replace function name
- **api_doc** (str, optional) – Docstring for API function
- ****kwargs** – Additional keyword arguments for handler class. Please reference to what arguments are available for the particular handler

Raises ValueError – API name must contains only letters

```
>>> from bentoml import BentoService, api
>>> from bentoml.handlers import JsonHandler, DataframeHandler
>>>
>>> class FraudDetectionAndIdentityService(BentoService):
>>>
>>>     @api(JsonHandler)
>>>     def fraud_detect(self, parsed_json):
>>>         # do something
>>>
>>>     @api(DataframeHandler, input_json_orient='records')
>>>     def identity(self, df):
>>>         # do something
```

3.2.1.3 env

bentoml.**env**(***kwargs*)

Define environment spec for BentoService

Parameters

- **setup_sh** (*str*) – User defined shell script to run before running BentoService. It could be local file path or the shell script content.
- **requirements_text** (*str*) – User defined requirement text to install before running BentoService.
- **pip_dependencies** (*str or list(str)*) – User defined python modules to install.
- **conda_channels** (*list(str)*) – User defined conda channels
- **conda_dependencies** (*list(str)*) – Defined dependencies to be installed with conda environment.
- **conda_pip_dependencies** (*list(str)*) – Additional pip modules to be install with conda

3.2.1.4 artifacts

bentoml.**artifacts**(*artifact_specs*)

Define artifact spec for BentoService

Parameters

- **artifact_specs** (*list(bentoml.artifact.ArtifactSpec)*) – A list of desired artifacts for initializing this BentoService
- **initializing this BentoService being decorated** (*for*) –

3.2.1.5 ver

bentoml.**ver**(*major*, *minor*)

Decorator for specifying the version of a custom BentoService.

Parameters

- **major** (*int*) – Major version number for Bento Service
- **minor** (*int*) – Minor version number for Bento Service

BentoML uses semantic versioning for BentoService distribution:

- MAJOR is incremented when you make breaking API changes
- MINOR is incremented when you add new functionality without breaking the existing API or functionality
- PATCH is incremented when you make backwards-compatible bug fixes

'Patch' is provided(or auto generated) when calling BentoService#save, while 'Major' and 'Minor' can be defined with '@ver' decorator

```
>>>  @ver(major=1, minor=4)
>>>  @artifacts([PickleArtifact('model')])
>>>  class MyMLService(BentoService):
>>>     pass
```

(continues on next page)

(continued from previous page)

```
>>>
>>>  svc = MyMLService.pack(model="my ML model object")
>>>  svc.save('/path_to_archive', version="2019-08.iteration20")
>>>  # The final produced BentoArchive version will be "1.4.2019-08.iteration20"
```

3.2.1.6 save

bentoml.**save**(*bento_service*, *dst*, *version=None*)

Save given BentoService along with all its artifacts, source code and dependencies to target path

Parameters

- **bento_service** (bentoml.service.BentoService) – a Bento Service instance
- **dst** (*str*) – Destination of where the bento service will be saved. It could be a local file path or a s3 path
- **version** (str, optional) – version text to use for saved archive

Returns The complete path of saved Bento service.

Return type string

3.2.1.7 load

bentoml.**load**(*archive_path*)

Load bento service from local file path or s3 path

Parameters **archive_path** (*str*) – The path that contains archived bento service. It could be local file path or aws s3 path

Returns The loaded bento service.

Return type *bentoml.service.BentoService*

3.2.1.8 config

bentoml.**config**()

BentoML configuration parser

:param default_config string - serve as default value when conf key not presented in environment var or user local config file

3.2.2 Artifacts

3.2.2.1 PickleArtifact

class bentoml.artifact.**PickleArtifact**(*name*, *pickle_module=<module 'dill' from '/home/docs/checkouts/readthedocs.org/user_builds/bentoml/envs/latest/lib/py packages/dill/__init__.py'>*, *pickle_extension='.pkl'*)

Abstraction for saving/loading python objects with pickle serialization

Parameters

- **name** (*str*) – Name for the artifact

- **pickle_module** (*module | str*) – The python module will be used for pickle and unpickle artifact
- **pickle_extension** (*str*) – The extension format for pickled file.

3.2.2.2 TextFileArtifact

class bentoml.artifact.**TextFileArtifact**(*name*, *file_extension='.txt'*, *encoding='utf8'*)
Abstraction for saving/loading string to/from text files

Parameters

- **name** (*str*) – Name of the artifact
- **file_extension** (str, optional) – The file extention used for the saved text file. Defaults to ".txt"
- **encoding** (*str*) – The encoding will be used for saving/loading text. Defaults to "utf8"

3.2.2.3 PytorchModelArtifact

class bentoml.artifact.**PytorchModelArtifact**(*name*, *file_extension='.pt'*)
Abstraction for saving/loading objects with torch.save and torch.load

3.2.2.4 XgboostModelArtifact

class bentoml.artifact.**XgboostModelArtifact**(*name*, *model_extension='.model'*)
Abstraction for save/load object with Xgboost.

Parameters

- **name** (*string*) – name of the artifact
- **model_extension** (*string*) – Extension name for saved xgboost model

Raises ImportError – xgboost package is required for using XgboostModelArtifact

3.2.2.5 FastaiModelArtifact

class bentoml.artifact.**FastaiModelArtifact**(*name*)
Saving and Loading FastAI Model

Parameters **name** (*str*) – Name for the fastai model

Raises

- ImportError – Require fastai package to use Fast ai model artifact
- ValueError – Model is not instance of fast ai model

3.2.2.6 H2oModelArtifact

class bentoml.artifact.**H2oModelArtifact**(*name*)
Abstraction for saving/loading objects with h2o.save_model and h2o.load_model

Parameters **name** (*str*) – Name for this h2o artifact..

Raises ImportError – h2o package is required to use H2o model artifact

3.2.2.7 KerasModelArtifact

class bentoml.artifact.**KerasModelArtifact**(*name*, *custom_objects=None*, *model_extension='.h5'*)

Abstraction for saving/loading Keras model

3.2.3 Handlers

3.2.3.1 DataframeHandler

class bentoml.handlers.**DataframeHandler**(*orient='records'*, *output_orient='records'*, *typ='frame'*, *input_dtypes=None*)

Dataframe handler expects inputs from rest request or cli options that can be converted into a pandas Dataframe, and pass down the dataframe to user defined API function. It also returns response for REST API call or print result for CLI call

Parameters

- **orient** (*str*) – Incoming json orient format for reading json data. Default is records.
- **output_orient** (*str*) – Prefer json orient format for output result. Default is records.
- **typ** (*str*) – Type of object to recover for read json with pandas. Default is frame
- **({str** (*input_dtypes*) – str}): A dict of column name and data type.

Raises

- ValueError – Incoming data is missing required columns in input_dtypes
- ValueError – Incoming data format is not handled. Only json and csv

3.2.3.2 ImageHandler

class bentoml.handlers.**ImageHandler**(*input_name='image'*, *accept_file_extensions=None*, *accept_multiple_files=False*, *pilmode='RGB'*)

Transform incoming image data from http request, cli or lambda event into numpy array.

Handle incoming image data from different sources, transform them into numpy array and pass down to user defined API functions

Parameters

- **input_name** (*string[]*) – A list of acceptable input name for HTTP request. Default value is image
- **accept_file_extensions** (*string[]*) – A list of acceptable image extensions. Default value is [.jpg, .jpeg, .png]
- **accept_multiple_files** (*boolean*) – Accept multiple files in single request or not. Default value is False
- **pilmode** (*string*) – The pilmode to be used for reading image file into numpy array. Default value is RGB. Find more information at https://imageio.readthedocs.io/en/stable/format_png-pil.html#png-pil

Raises ImportError – imageio package is required to use ImageHandler

3.2.3.3 FastaiImageHandler

class bentoml.handlers.**FastaiImageHandler**(*input_name=None*, *accept_file_extensions=None*, *convert_mode=None*, *div=True*, *cls=None*, *after_open=None*)

Transform incoming image data to fastai.vision.Image

Handle incoming image data, process them into fastai.vision.Image instance and pass down to user defined API functions

Parameters

- **input_name** (*[str]*) – A list of acceptable input name for HTTP request. Default value is image
- **accept_file_extensions** (*[str]*) – A list of acceptable image extensions. Default value is [.jpg, .jpeg, .png]
- **accept_multiple_files** (*boolean*) – Accept multiple files in single request or not. Default value is False
- **convert_mode** (*str*) – The pilmode to be used for reading image file into numpy array. Default value is RGB. Find more information at https://imageio.readthedocs.io/en/stable/format_png-pil.html#png-pil
- **div** (*bool*) – If True, pixel values are divided by 255 to become floats between 0. and 1.
- **cls** (*Class*) – Parameter from fastai.vision `open_image`, default is `fastai.vision.Image`
- **after_open** (*func*) – Parameter from fastai.vision `open_image`, default is None

Raises

- `ImportError` – imageio package is required to use FastaiImageHandler
- `ImportError` – fastai package is required to use FastaiImageHandler

3.2.3.4 JsonHandler

class bentoml.handlers.**JsonHandler**

JsonHandler parses REST API request or CLI command into parsed_json(a dict in python) and pass down to user defined API function

3.2.4 Deployments

3.2.4.1 deploy_bentoml

bentoml.deployment.clipper.**deploy_bentoml**(*clipper_conn, archive_path, api_name, input_type='strings', model_name=None, labels=['bentoml']*)

Deploy bentoml bundle to clipper cluster

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection*) – Clipper connection instance
- **archive_path** (*str*) – Path to the bentoml service archive.

- **api_name** (*str*) – name of the api that will be used as prediction function for clipper cluster
- **input_type** (*str*) – Input type that clipper accept. The default input_type for image handler is *bytes*, for other handlers is *strings*. Availabel input_type are *integers*, *floats*, *doubles*, *bytes*, or *strings*
- **model_name** (*str*) – Model's name for clipper cluster
- **labels** (list(str), optional) – labels for clipper model

Returns Model name and model version that deployed to clipper

Return type tuple

3.2.5 Exceptions

3.2.5.1 BentoMLException

exception bentoml.exceptions.**BentoMLException**
Base class for all BentoML's errors. Each custom exception should be derived from this class

3.3 CLI

3.3.1 bentoml

BentoML CLI tool

```
bentoml [OPTIONS] COMMAND [ARGS]...
```

Options

-q, --quiet
Hide process logs and only print command results

--verbose
Print verbose debugging information for BentoML developer

--version
Show the version and exit.

3.3.1.1 <API_NAME>

Run a API defined in saved BentoArchive with cli args as input

```
bentoml <API_NAME> [OPTIONS] API_NAME ARCHIVE_PATH
```

Options

--with-conda
Run API server in a BentoML managed Conda environment

Arguments

API_NAME
Required argument

ARCHIVE_PATH
Required argument

3.3.1.2 check-deployment-status

Check deployment status of BentoML archive

```
bentoml check-deployment-status [OPTIONS] ARCHIVE_PATH
```

Options

--platform <platform>
Target platform that Bento archive will be deployed to as a REST api service [required]

Options aws-lambda|aws-lambda-py2|gcp-function|aws-sagemaker|azure-ml|algorithmia

--region <region>
Deployment's region name inside cloud provider. [required]

--stage <stage>

--api-name <api_name>
The name of API that is deployed as a service.

Arguments

ARCHIVE_PATH
Required argument

3.3.1.3 config

Configure BentoML configurations and settings

```
bentoml config [OPTIONS] COMMAND [ARGS]...
```

reset

Reset BentoML configuration to default

```
bentoml config reset [OPTIONS]
```

set

Set value to BentoML configuration

```
bentoml config set [OPTIONS] [UPDATES]...
```

Arguments

UPDATES
Optional argument(s)

unset

Unset value from BentoML configuration

```
bentoml config unset [OPTIONS] [UPDATES]...
```

Arguments

UPDATES
Optional argument(s)

view

View BentoML configurations

```
bentoml config view [OPTIONS]
```

view-effective

```
bentoml config view-effective [OPTIONS]
```

3.3.1.4 delete-deployment

Delete active BentoML deployment from cloud services

```
bentoml delete-deployment [OPTIONS] ARCHIVE_PATH
```

Options

--platform <platform>
The platform bento archive is deployed to [required]

Options aws-lambda|aws-lambda-py2|gcp-function|aws-sagemaker|azure-ml|algorithmia

--region <region>
The region deployment belongs to [required]

--api-name <api_name>
Name of the API function that is deployed

--stage <stage>

Arguments

ARCHIVE_PATH
Required argument

3.3.1.5 deploy

Deploy BentoML archive as REST endpoint to cloud services

```
bentoml deploy [OPTIONS] ARCHIVE_PATH
```

Options

--platform <platform>
Target platform that Bento archive is going to deployed to [required]

Options aws-lambda|aws-lambda-py2|gcp-function|aws-sagemaker|azure-ml|algorithmia

--region <region>
Target region inside the cloud provider that will be deployed to

--stage <stage>

--api-name <api_name>
The name of API will be deployed

--instance-type <instance_type>
SageMaker deployment ONLY. The instance type to use for deployment

--instance-count <instance_count>
Sagemaker deployment ONLY. Number of instances to use for deployment

Arguments

ARCHIVE_PATH
Required argument

3.3.1.6 docs

Display API documents in Open API format

```
bentoml docs [OPTIONS] ARCHIVE_PATH
```

Arguments

ARCHIVE_PATH
Required argument

3.3.1.7 info

List all APIs defined in the BentoService loaded from archive.

```
bentoml info [OPTIONS] ARCHIVE_PATH
```

Arguments

ARCHIVE_PATH
Required argument

3.3.1.8 serve

Start REST API server hosting BentoService loaded from archive

```
bentoml serve [OPTIONS] ARCHIVE_PATH
```

Options

--port <port>
The port to listen on for the REST api server, default is 5000.

--with-conda
Run API server in a BentoML managed Conda environment

Arguments

ARCHIVE_PATH
Required argument

3.3.1.9 serve-gunicorn

Start REST API gunicorn server hosting BentoService loaded from archive

```
bentoml serve-gunicorn [OPTIONS] ARCHIVE_PATH
```

Options

-p, --port <port>

-w, --workers <workers>
Number of workers will start for the gunicorn server

--timeout <timeout>

--with-conda
Run API server in a BentoML managed Conda environment

Arguments

ARCHIVE_PATH
Required argument

3.4 Deployments

3.4.1 Overview

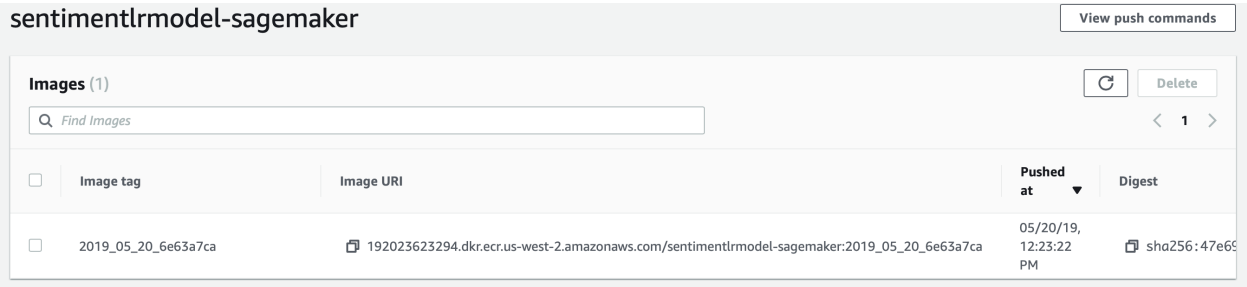
BentoML helps deploy bento archive as REST API server to cloud services. Using the `bentoml deploy` command, we can easily deploy to different cloud services.

3.4.2 Deploying to AWS Sagemaker

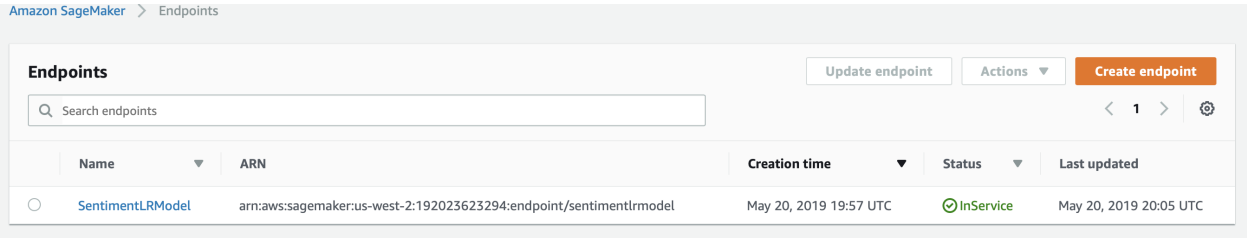
AWS Sagemaker is a service for training and serving machine learning models. With BentoML, you can deploy any custom model to Sagemaker without writing additional code.

```
$ bentoml deploy my/bento_archive_path/sentimentLR --platform aws-sagemaker --region␣
↪us-west2
```

BentoML will build docker image compatible with Sagemaker locally and push the image to AWS ECR (Elastic container registry).



After pushing the image to ECR. BentoML will create configuration and model for Sagemaker and create endpoint afterward.



Once the endpoint is *InService*, we can make request against it.

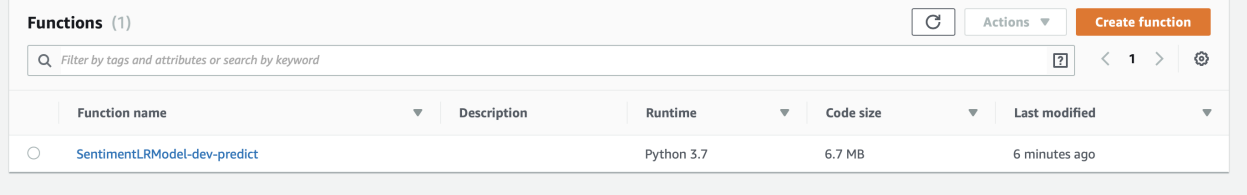
```
bozhaoyu@Bozhaos-MacBook-Pro ~/src/bento  sagemaker-doc-example ● ? ⊛7
└─ aws sagemaker-runtime invoke-endpoint \
--endpoint-name SentimentLRModel \
--body '["new food", "bad movie", "chicken nuggets", "good family time"]' \
--content-type "application/json" \
output.json
{
    "ContentType": "application/json",
    "InvokedProductionVariant": "SentimentLRModel"
}
bozhaoyu@Bozhaos-MacBook-Pro ~/src/bento  sagemaker-doc-example ● ? ⊛7
└─ cat output.json
[4, 0, 0, 4]%
```

3.4.3 Deploying to AWS Lambda

Lambda is a serverless service offer by AWS. Serverless services offer abilities to scale automatically base on demands, and reduce administrative operation tasks for users. Deploy to AWS Lambda use the same `bentoml deploy` command. For platform, you can choose between `aws-lambda` for python 3 project or `aws-lambda-py2` for python 2 project.

```
$ bentoml deploy my/bento_archive_path/sentimentLR --platform aws-lambda --region us-
↪west2
```

After successful deployment, the deployed REST api service will show up in the AWS lambda's dashboard as a function.



We can make HTTP POST request to the function and get result back from it.

```
└─ curl -i \
--header "Content-Type: application/json" \
--request POST \
--data '["some new text, sweet noodles", "happy time", "sad day"]' \
https://dhy292o6r6.execute-api.us-west-2.amazonaws.com/dev/predict
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 9
Connection: keep-alive
Date: Fri, 24 May 2019 20:31:58 GMT
x-amzn-RequestId: fa7ad7c3-7e62-11e9-b4af-a3ad35996092
x-amz-apigw-id: aNItxGD5PHcFX0A=
X-Amzn-Trace-Id: Root=1-5ce854be-68be202317c4334552c85a55;Sampled=0
X-Cache: Miss from cloudfront
Via: 1.1 d2bd759914e30b1d5aee2929535c55f9.cloudfront.net (CloudFront)
X-Amz-Cf-Id: aCXn-LTBeVSg8iqLEUfDL8CbxgbWpnp8mBfhjhshoQiT8_HbAwMg8w==

[4, 4, 0]%
```

3.4.4 Deploying with Docker

```
$ cd my/bento_archive_path/irisclassifier
$ docker build . -t iris-classifier-docker-tag
```

Now we have the docker image, we can push it to our docker image registry, and from there we can deploy to AWS fargate or any kubernetes cluster.

3.4.5 Deploying with Kubernetes

Kubernetes is a popular container orchestration system that companies are using for operation their applications and services. It can run on-prem or any cloud providers. Since kubernetes can be highly customizable to work with different configuration and needs, we will provide an simple example for it.

After building the docker image and pushed it to a docker image registry, we can use that image to deploy a service to kubernetes cluster. After applying the configurations to the kubernetes cluster, It will create a service and expose an endpoint for us to make request with.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: iris-classifier
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      name: iris-classifier
  template:
    metadata:
      labels:
        name: iris-classifier
    spec:
      containers:
        - name: iris-classifier
          image: iris-classifier-docker-tag
          imagePullPolicy: Always
          ports:
            - containerPort: 5000
              protocol: TCP

------------

apiVersion: v1
kind: Service
metadata:
  name:iris-classifier
  namespace: default
  labels:
    name: iris-classifier
    app: iris-classifier
spec:
  ports:
    - port: 80
      name: http
      protocol: TCP
```

(continues on next page)

(continued from previous page)

```
    targetPort: 5000
  selector:
    name: iris-classifier
  type: NodePort
```

3.5 Using Bento Archive

There are few ways to utilize Bento archive to work with different serving scenarios. We will list out few ways to use Bento archive and the serving scenarios that they would be great fit with.

3.5.1 Using inside python application

It is easy to use Bento archive with python application. Import BentoML package into your application and load archive with `bentoml.load` function

Load function works with local file system as well as AWS S3 path. You can find more information at API reference page *load* section.

```
import bentoml

bento_service = bentoml.load('my/bento_archive/path/IrisClassifier')

api_list = bento_service.get_service_apis()

# the get_service_apis function will return a list of user defined APIs.
print(api_list)

print(bento_service.predict(INPUT_DATA))
```

In the example above, we imported bentoml and use load function to load our IrisClassifier archive. After we load the archive into a bento service, we can make prediction with `bento_service.predict`.

If you defined more than one API functions, you can get the list of defined API functions with `get_service_apis` method that's availabel on bento service.

3.5.2 Install with `pip install`

Bento archive includes a generated `setup.py` file that we can use to distribute and use the archive as PyPi package.

Just navigate to the archive's directory and run `pip install .`

```
$ cd my/bento_archive/path/IrisClassifier
$ pip install .
```

We can also publish this archive as python package to pypi.org or private PyPi index, after you configured your `.pypirc` file.

```
$ cd my/bento_archive/path/IrisClassifier
$ python setup.py sdist upload
```

We can import and use bento archive just like a normal python package after `pip install`. After you import package, you will need to call `load` function to initialize it into a bento service.

```
import IrisClassifier

service = IrisClassifier.load()

print(service.predict(INPUT_DATA))
```

3.5.2.1 Using generated CLI tool

Another benefit of using `pip install` approach is the generated CLI tool. Bento archive created a customized CLI tool for you.

You can check API info by calling *info* command.

```
$ IrisClassifier info
```

Make prediction with data from local file or string.

```
$ IrisClassifier predict --input JSON/FILE/PATH
$ IrisClassifier predict --input '{"key": "value", "json": "string"}'
```

Start a local REST API server with *serve* command. Default port is 5000, use `--port` options to change.

```
$ IrisClassifier serve --port 5001
```

3.5.3 Using BentoML CLI tool

You can use BentoML's CLI tool without install the archive with pip. Just provide the archive's location as local path or s3 location.

```
$ bentoml info my/bento_archive/IrisClassifier
$ bentoml info s3://my_bucket/bento_archive_path/IrisClassifier
```

```
$ bentoml predict my/bento_archive/IrisClassifier --input JSON/FILE/PATH
```

```
$ bentoml serve my/bento_archive/IrisClassifier --port 5001
```

We can start a gunicorn server to maximize on utilizing our computing resources.

```
$ bentoml serve-gunicorn my/bento_archive/IrisClassifier --port 5001 --workers 2
```

3.5.4 Generate Docker Image

BentoML also support to build docker image with bento archive. Docker is one the most common packaging format for deploying applications. With bento archive, navigate to the archive directory and run `docker build`

```
$ cd my/bento_archive/IrisClassifier
$ docker build . -t iris-classifier
```

After finish building docker image, we can run the image with `docker run`

```
$ docker run -p 5000:5000 iris-classifier
```

Index

Symbols

A

T

U

V

X