Clipper Documentation

Release 0.2.0rc1

Dan Crankshaw

Jun 06, 2018

Contents

CHAPTER 1

Clipper Connection

`ClipperConnection` is the primary way of starting and managing a Clipper cluster.

CHAPTER 2

Container Managers

Container managers abstract away the low-level tasks of creating and managing Clipper's Docker containers in order to allow Clipper to be deployed with different container orchestration frameworks. Clipper currently provides two container manager implementations, one for running Clipper locally directly on Docker, and one for running Clipper on Kubernetes.

CHAPTER 3

Model Deployers

Clipper provides a collection of model deployer modules to simplify the process of deploying a trained model to Clipper and avoid the need to figure out how to save models and build custom Docker containers capable of serving the saved models for some common use cases. With these modules, you can deploy models directly from Python to Clipper.

Currently, Clipper provides two deployer modules, one to deploy arbitrary Python functions (within some constraints) and the other to deploy PySpark models along with pre- and post-processing logic.

Note: You can find additional examples of using both model deployers in Clipper's integration tests.

3.1 Pure Python functions

This module supports deploying pure Python function closures to Clipper. A function deployed with this module must take a list of inputs as the sole argument, and return a list of strings of exactly the same length. The reason the prediction function takes a list of inputs rather than a single input is to provide models the possibility of computing multiple predictions in parallel to improve model performance. For example, many models that run on a GPU can significantly improve throughput by batching predictions to better utilize the many parallel cores of the GPU.

In addition, the function must only use pure Python code. More specifically, all of the state captured by the function will be pickled using Cloudpickle, so any state captured by the function must be able to be pickled. Most Python libraries that use C extensions create objects that cannot be pickled. This includes many common machine-learning frameworks such as PySpark, TensorFlow, PyTorch, and Caffe. You will have to create your own Docker containers and call the native serialization libraries of these frameworks in order to deploy them.

The base Docker image we use to deploy these functions has already installed several common Python dependencies (the dependencies are specified in the `python_container_conda_deps.txt` file included in this package). In addition, this module will attempt to capture any additional Python modules your function uses. Additional modules that were installed with Anaconda or Pip will be installed in the same way in your deployed Docker container when the container is started. Additional local modules will be copied to the container along with the function.

While Clipper will try to capture additional dependencies and install them, these dependencies will not be built into the Docker image but instead will be downloaded and installed each time a container is started. As a result, this can significantly increase container initialization time (sometimes taking several minutes to start up). To remedy this, you can create a new Docker image based on the default image `clipper/python-closure-container:0.2` that installs these additional dependencies.

For example, if you knew your function needed the Python package `networkx`, you could create the following Dockerfile:

```
FROM clipper/python-closure-container:0.2
RUN pip install networkx
```

You would then use this Dockerfile to build a new image:

```
docker build -t python-closure-with-networkx .
```

Then, when you call `deploy_python_closure` or `create_endpoint`, instead of leaving the `base_image` argument to be the default, you would set it to `base_image="python-closure-with-networkx"` and now the container won't need to install networkx each time th container is initialized.

Warning: Dependency-Capture Caveats: All of the dependency capture is performed on a best-effort basis and you may run into edge-cases that Clipper cannot support. Furthermore, the dependency-capture logic relies on being called from inside an Anaconda environment, so if you call this function from outside an Anaconda environment no dependency capture will be attempted.*

3.2 PySpark Models

The PySpark model deployer module provides a small extension to the Python closure deployer to allow you to deploy Python functions that include PySpark models as part of the state. PySpark models cannot be pickled and so they break the Python closure deployer. Instead, they must be saved using the native PySpark save and load APIs. To get around this limitation, the PySpark model deployer introduces two changes to the Python closure deployer discussed above.

First, a function deployed with this module *takes two additional arguments*: a PySpark `SparkSession` object and a PySpark model object, along with a list of inputs as provided to the Python closures in the `deployers.python` module. It must still return a list of strings of the same length as the list of inputs.

Second, the `pyspark.deploy_pyspark_model` and `pyspark.create_endpoint` deployment methods introduce two additional arguments:

- `pyspark_model`: A PySpark model object. This model will be serialized using the native PySpark serialization API and loaded into the deployed model container. The model container creates a long-lived SparkSession when it is first initialized and uses that to load this model once at initialization time. The long-lived SparkSession and loaded model are provided by the container as arguments to the prediction function each time the model container receives a new prediction request.
- `sc`: The current SparkContext. The PySpark model serialization API requires the SparkContext as an argument

The effect of these two changes is to allow the deployed prediction function to capture all pure Python state through closure capture but explicitly declare the additional PySpark state which must be saved and loaded through a separate process.

All caveats about dependency-capture still hold.

CHAPTER 4

Exceptions