# Implementing ML for Cortex

**Contents:**

Cortex helps you to manage the lifecycle, execution, versioning, and data storage of Machine Learning (ML) implementations. ML implementations are incorporated into Cortex as Skills, which can be added to one or more agents. To access an ML algorithm from a Cortex skill, it must be packaged up in a Docker container and deployed to a location that Cortex can access.

Understanding who is involved in ML implementations development can help to distinguish roles and responsibilities when planning a Cortex project. These responsibilities can be spread out across a team or be fulfilled by the same person as appropriate for your requirements.

- The **Data Scientist** is responsible for writing the ML implementation and packaging it as a Docker image.
- The **Cortex Skill Developer** is responsible for creating a Cortex skill that uses the ML implementation packaged by the Data Scientist.

**This guide describes the steps for the Data Scientist.** To help you understand the process, we walk you through the steps using a concrete example: a Real Estate mean value estimator that uses a Random Forest algorithm.

## Note

This guide assumes that you are working with a pre-existing algorithm, either one that you have developed yourself or one that is accessible from an ML framework.

## Glossary

- **Model Implementation**: The concrete, runnable implementation of an ML algorithm.
- **Trained Model**: A serialized piece of code that results from training via a Model Implementation.
- **Containerized Model**: A *Model Implementation* that has been packaged in a Docker image.
- **Model Event**: An arbitrary piece of information related to a *Model*.
- **Inquiry**: A request to a *trained model* for an answer.
- **Model Server**: A web server (in Docker) that responds to inquiry requests by using the *trained model*.

## 1 - Install the library

Use the Cortex Python Library to integrate your ML algorithm with Cortex. It is publicly available from pypi.python.org. Install the library with pip.

Copy

```
pip install cortex-client
```

## 2 - Create an ML project

To get started, create an ML project with the following directory structure.

Copy

```
.
├── Dockerfile
└── model
    ├── main.py
    └── model.py
```

The sections that follow describe what to include in each file.

## 3 - Define your model

it. By implementing this interface, Cortex is able to execute any ML implementation.

With the `ModelProcess` interface, the Data Scientist specifies:

1. How a Model should be trained.
2. How an inquiry (inference) request should be handled.

The Data Scientist does not need to know or worry about:

- Where the trained model is stored to or retrieved from.
- How the train and inquire requests are run and served.

### Implement the `train` method

The `train` method of the `ModelProcess` has the following signature:

Copy
```
@staticmethod
@abc.abstractmethod
def train(request: Dict[str, object],
          cortex_model: Model,
          datasets_client: DatasetsClient,
          client: ModelClient) -> None:
    """Perform training of the Model.

    :param request: The arguments needed to train the model.
    :param cortex_model: The Cortex Model object with which this train is
    :param datasets_client: The DatasetsClient to download training data.
    :param model_client: The ModelClient with methods to report training events.
    """
```

The example below shows an implementation of the `train` method for a Real Estate example.

Copy
```
@staticmethod
def train(request, cortex_model, datasets_client, model_client):
    data = RealestateRF._fetch_training_data(request, datasets_client)
    X = data.data
    y = data.target
    # Split dataset into train and test/validation sets:
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    trained_model = RandomForestRegressor(
                n_estimators=request['args']['n_estimators'],
                min_samples_leaf=request['args']['min_samples_leaf'],
                random_state=1)
    # Fit the model to the data.
    trained_model.fit(X_train, y_train)

    validation = trained_model.score(X_test, y_test)
    model_client.log_event(cortex_model, 'train.validation', validation)

    # save
    b = RealestateRF._serialize("serialized_model", trained_model)
    model_client.save_state(cortex_model, "serialized_model", b)

@staticmethod
def _fetch_training_data(request, datasets_client):
    data = datasets_client.get_dataframe(request['dataset'])
    data2 = numpy.array(data['values']).astype(numpy.float)
    X = data2[:,0:-1]
    y = data2[:,-1]
    return namedtuple("Data", ("data", "target"))(X, y)
```

*Explanation:*

- The `request` contains the training parameters sent in the train request. In this case, the algorithm expects an `args` key with two params: `n_estimators` and `min_samples_leaf`.
- The `datasets_client` has methods to download data (e.g., training data) stored in Cortex.
- `model_client` is used to save the trained model and, optionally, to report execution events.
- `cortex_model` is only needed to report events related to this train request back to Cortex. In this example we report a "`train.validation`" event with the value of the `trained_model.score()` call.
- Finally, the function saves the trained model for subsequent use on model serving.

### Implement the `inquire` method

The `inquire` method of the `ModelProcess` has the following signature:

```
def inquire(request: Dict[str, object],
            cortex_model: Model,
            model_client: ModelClient) -> JSONType:
    """Performs ask against a `trained_model`.

    :param request: The arguments from the end user to perform inquiry.
    :param cortex_model: The Cortex Model object with which this train is
        associated. This sould be used with the ModelClient to log Model events.
    :param model_client: The ModelClient with methods to log training events.

    :return: The inquiry result.
    """
```

The example below shows an implementation of the `inquire` method for the Real Estate example.

```
@staticmethod
def inquire(request, cortex_model, model_client):
    ser_model = model_client.load_state(cortex_model, "serialized_model").read()
    trained_model = RealestateRF._deserialize('serialized_model', ser_model)
    return trained_model.predict(request['args'])
```

*Explanation:*

- `request` again contains the inquiry request arguments.
- `model_client` is used to load the trained model.
- `cortex_model` is again an object needed for reporting execution events. It is not used here.

## Name your ML implementation

Finally, the `ModelProcess` has a `name` attribute to specify the name of the Model. Cortex uses this to identify the Model (along with Agent and Skill attributes).

```
class RealestateRF(ModelProcess):

    name = 'RealestateRF'
```

## 4 - Define the model's entry point

The `main.py` file is the entry point to execute the Docker image. The `main.py` script must link your `ModelProcessor` implementation from `model.py` with Cortex's `ModelRunner` and `ModelRouter` . By passing an instance of the `ModelProcessor` implementation to these classes, you can connect an ML implementation with Cortex.

The following sections provide an overview of the `ModelRunner` and `ModelRouter` . In addition, an example implementation is provided for the Real Estate example.

### ModelRunner

The `ModelRunner` is responsible for executing the `ModelProcessor.train()` and `ModelProcessor.inquire()` functions implemented by the Data Scientist.

On `train` , the `ModelRunner` :

- Creates a new Model version that uniquely identifies a train request, and that serves as an identifier linking all artifacts (Model Events, serialized models) resulting from the train (and subsequent related inquiry) execution.
- Calls the `ModelProcessor.train()` function implemented by the Data Scientist.
- Reports execution events related to the train request.

On `inquiry` , the `ModelRunner` :

- Retrieves the serialized model needed to serve the request.
- Calls `ModelProcess.inquire()` implemented by the Data Scientist.
- Reports execution events relevant for the inquiry request.

### ModelRouter

The `ModelRouter` implements a CLI (Command Line Interface) for running the ML implementation. It defines three CLI commands:

- `--train` for executing a train request.
- `--inquire` for executing an inquiry request.

The `--train` and `--inquire` commands require a `--context` argument with a Cortex `InputMessage` as JSON payload. Data Scientists implementing the `ModelProcessor` interface need not know about this CLI or the `InputMessage` except for local testing, because the `ModelProcessor` is agnostic to it.

### Example implementation

The example below shows how to implement the `ModelRunner` and `ModelRouter` for the Real Estate example.

Copy

```
from cortex_client.webserver import webserver_app
from cortex_client import ModelRouter
from cortex_client import ModelRunner

from model import RealestateRF

webserver_app.modelrunner = ModelRunner(RealestateRF())

if __name__ == '__main__':
    ModelRouter.main(RealestateRF())
```

## 5 - Define the project's Dockerfile

The `Dockerfile` is a specification that defines how to build the Docker image. For help with Docker, refer to their documentation.

For an ML implementation in Cortex that supports training and serving inquiries, the `Dockerfile` must specify the following:

- The dependencies to serve inquiry requests (Flask, Nginx, Gunicorn)
- The Cortex Python Library
- The `model` folder with your ML implementation
- The dependencies for your own ML implementation (e.g., numpy, scipy)

The `ENTRYPOINT` to your Docker image should be `["python", "<path-to-model-dir>/main.py"]`

If you use the Cortex base image, your `Dockerfile` only needs to:

- Copy your `model` directory to `/opt/program`.
- Install dependencies for your ML.

In that case, your `Dockerfile` can be as simple as:

```
FROM c12e/cortex-python-lib
MAINTAINER CognitiveScale.com

COPY model /opt/program
```

After building your Docker image, upload it to your Docker repo. As long the repository is public or managed by Cortex, Cortex can now use it.

> **Note**
>
> Cortex does not currently support accessing private repositories. This functionality will be available soon.

## Next steps

- Walk through the [Hello World for Machine Learning][Link-hello-world-ml] tutorial to learn how skill developers can create a skill that uses an ML implementation like the one described above.

**On this page**

Glossary