modelstore

Neal Lathia

Dec 20, 2022

BASIC USAGE

`modelstore` is a machine learning model registry Python library.

By saving your models using `modelstore`, you can:

- Version your models;
- Upload model artefacts to your choice of storage;
- Collect meta data about the models your uploading;
- Control models' states;
- Load models straight from storage back into memory

CHAPTER

ONE

INSTALLING THE MODELSTORE LIBRARY

This library can be installed via pip:

```
pip install modelstore
```

You can find the latest version here: modelstore on Pypi.

You can also build the library directly from source by checking it out from Github.

CHAPTER

TWO

QUICK START

2.1 Install using pip, import in your code

The model store library is available via Pypi:

```
pip install modelstore
```

In your code, import `ModelStore` with:

```
from modelstore import ModelStore
```

2.2 Create a model store instance and point it to your storage

The model store library supports storing models to blob storage across different cloud providers:

- A file system;
- Google Cloud storage buckets
- AWS s3 buckets
- Azure blob storage containers
- MinIO object storage

Create a model store instance by using one of the following factory methods.

File System Storage

```
model_store = ModelStore.from_file_system(root_directory="/path/to/directory")
```

Google Cloud Storage Bucket

```
model_store = ModelStore.from_gcloud(
   project_name="my-project",
   bucket_name="my-bucket",
)
```

AWS s3 Bucket

```
model_store = ModelStore.from_aws_s3(
   bucket_name="my-bucket",
)
```

Azure Blob Storage

```
model_store = ModelStore.from_azure(container_name="my-container-name")
```

2.3 Upload a model to the model store

Model store has an `upload()` function that will create an archive containing your model and upload it to your storage.

Whenever you upload a model, you need to specify which domain it belongs to. A "domain" is a string that model store uses to group several models that are for the same end-usage together.

For example, let's say you've trained a scikit-learn model (which is stored in a variable called `clf`) that is going to be used in a spam classifier domain.

To store the model, use:

```
meta_data = model_store.upload("spam-detection", model=clf)
```

The `upload()` function returns a dictionary containing meta data about your model - including the id that has been assigned to it, which is in `meta_data["model"]["model_id"]`.

2.4 Load a model from the model store

Once a model has been stored, you can load it straight from storage back into memory using model store's `load()` function.

```
clf = model_store.load("spam-detection", model_id="abcd-abcd-abdc")
```

CHAPTER

THREE

UPLOADING A SCIKIT-LEARN MODEL

This example is based on the GradientBoostingRegressor tutorial from the scikit-learn website:

```python
import json
import os

from sklearn.datasets import load_diabetes
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split

from modelstore import ModelStore


def train():
    diabetes = load_diabetes()
    X_train, X_test, y_train, y_test = train_test_split(
        diabetes.data, diabetes.target, test_size=0.1, random_state=13
    )
    params = {
        "n_estimators": 500,
        "max_depth": 4,
        "min_samples_split": 5,
        "learning_rate": 0.01,
        "loss": "ls",
    }
    reg = GradientBoostingRegressor(**params)
    reg.fit(X_train, y_train)
    # Skipped for brevity (but important!) evaluate the model
    return reg


if __name__ == "__main__":
    # In this demo, we train a GradientBoostingRegressor
    # using the same approach described on the scikit-learn website.
    # Replace this with the code to train your own model
    model = train()

    # The modelstore library currently assumes you have already created
    # a Cloud Storage bucket and will raise an exception if it doesn't exist

    # This example assumes that you have the GCP project name and bucket id
```

(continues on next page)

(continued from previous page)

```
    # saved as environment variables - replace the os.environ below with
    # your values
    model_store = ModelStore.from_gcloud(
        project_name=os.environ["GCP_PROJECT_ID"],
        bucket_name=os.environ["GCP_BUCKET_NAME"],
    )

    # Upload the model
    meta_data = model_store.upload(
        "sklearn-diabetes-boosting-demo",
        model=model
    )

    # The upload returns meta-data about the model that was uploaded
    # This meta-data has also been sync'ed into the cloud storage
    # bucket
    print("  Finished uploading model!")
    print(json.dumps(meta_data, indent=4))

    # Download the model back!
    target = f"downloaded-{model_type}-model"
    os.makedirs(target, exist_ok=True)
    model_path = model_store.download(
        local_path=target,
        domain=model_domain,
        model_id=meta["model"]["model_id"],
    )
    print(f"  Downloaded the model back to {model_path}")
```

CHAPTER

FOUR

KEY CONCEPTS

The `modelstore` library is built around a few key concepts.

4.1 Model Archive

When you upload a model, an `artifacts.tar.gz` file is created and then uploaded to your storage. This archive contains:

1. Files that are dumps from your model,
2. A `"python-info.json"` file that enumerates the version of the Python library of the model you are exporting.

4.2 Model Meta-data

The `upload()` function returns a dictionary containing meta-data about the model, which includes an id for your model.

The meta-data includes:

- A unique id for your model;
- Details about where the model is being uploaded to (the bucket and prefix);
- The Python runtime that was used (e.g., "python:3.7.0")
- The user who ran the upload.
- Versions for the Python library and key dependencies.

4.3 Domains

A **domain** is how `modelstore` denotes a group of models, that are all intended for the same end-usage. When you upload a model to the store, you will add it to a domain.

The model store library then allows you to list the models that are in a domain and retrieve specific models (e.g., the latest one).

Under the hood, a domain is just a string, so it is up to you how you would like to use it.

4.4 Model State

A **model state** is a tag that you can use to control the lifecycle of a model in a given domain.

For example, you may want to have some models tagged as being in state "production" or state "shadow." You can achieve this by creating a state and then setting a model's state.

Under the hood, a model state name is just a string, so it is up to you how you would like to use it.

4.5 Storage

When you pick a backend that stores data in files (e.g., Cloud Storage Buckets), the files are stored with a pre-defined structure.

The top-level, **root** prefix that this library hard-codes is `operatorai-model-store`.

When you create and upload a model archive, this library will upload three files to different places in the bucket.

1. **The artifacts archive** will be uploaded to: `root/<domain>/<datetime>/archive.tar.gz`, where the datetime has the form `"%Y/%m/%d/%H:%M:%S"` - denoting the time when the model was uploaded.
2. The library creates a dictionary of **meta-data** about your model. This will be uploaded to `root/<domain>/versions/<model-id>.json`.
3. This same meta-data is also stored in `root/<domain>/latest.json`, which tracks the _last_ model that was uploaded to the model store.

CHAPTER

FIVE

SUPPORTED MACHINE LEARNING LIBRARIES

This library currently supports several different machine learning libraries. To save models trained with them, you should use the upload function:

```
model_store.upload("domain", <kwargs>)
```

Table 1: Supported machine learning libraries

Library	Required kwargs	Example code
Annoy	model	Annoy Example
CatBoost	model, pool (for classification)	Catboost Example
FastAI	learner	FastAI Example
Gensim	model	Word2vec Example
Keras	model, optimizer	Keras Example
LightGBM	model	LightGBM Example
Mxnet	model, epoch	Mxnet Example
Onnx	model	Onnx Example
Prophet	model	Prophet Example
PySpark ML Lib	model	PySpark Example
PyTorch	model, optimizer	PyTorch Example
PyTorch Lightning	model, trainer	PyTorch Lightning Example
Shap	explainer	Shap Example
scikit-learn	model	scikit-learn Example
skorch	model	skorch Example
Tensorflow	model	Tensorflow Example
Transformers	config, model, tokenizer	Transformers Example
XGBoost	model	XGBoost Example

5.1 What to do if a library is not supported

If you are using a machine learning library that is not listed above, you can still use model store to upload and version your models by uploading a file. You will not be able to use `load()` but you will be able to `download()` them back.

```
model_path = save_model()

model_store.upload("my-domain", model=model_path)
```

You can also:

- Let us know by raising an issue
- Add support for the library by following this guide.

CHAPTER

SIX

SUPPORTED STORAGE TYPES

This library currently supports several places where you can save your models. You specify the storage type when you create a ModelStore instance:

Table 1: Supported storage types

Storage	Requires	Example code
AWS s3	The name of an existing s3 bucket	AWS Example
MinIO s3 storage	The name of an existing bucket and access credentials	MinIO Example
Azure Container	The name of an existing container	Azure Example
Google Cloud Storage	The name of an existing bucket	Cloud Storage Example
File system	A path	File system Example

6.1 File system storage

The file system model storage assumes that (a) the root directory exists, and (b) the library user has permission to write to it.

If you want to create the root directory if it does not exist, pass along the *create_directory=True* argument.

```
model_store = ModelStore.from_file_system(
  root_directory="/path/to/directory",
  create_directory=True,
)
```

CHAPTER

SEVEN

ADDITIONAL UPLOAD FUNCTIONALITY

7.1 Uploading more than one model file

This library supports uploading multiple models, as long as their keyword arguments do not overlap.

For example, you might want to upload a classifier **and** a shap explainer together:

```
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

explainer = shap.TreeExplainer(model)

model_store.upload("my-domain", model=model, explainer=explainer)
```

When you load these models, model store returns a dictionary with both models:

```
models = modelstore.load(model_domain, model_id)
clf = models["sklearn"]
explainer = models["shap"]
```

7.2 Uploading extra files with the model

This library supports uploading a model with one or more extra files.

For example, you might want to upload a classifier **and** the predictions it made on the test set.

```
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

predictions = clf.predict(X_test)
file_path = "predictions.csv"
numpy.savetxt(file_path, predictions, delimiter=",")

modelstore.upload("my-domain", model=model, extras=file_path)
```

When you load these models, the extra files are not loaded into memory:

```
clf = modelstore.load(model_domain, model_id)
```

CHAPTER EIGHT

ADDITIONAL DOWNLOAD FUNCTIONALITY

8.1 Providing read-only access

The AWS s3, Google GCS, Azure Containers storage types assume that (a) the bucket/container exists, and (b) the library user has both read and write permissions.

As of 0.0.74, modelstore also supports read-only access to public Google Cloud Storage buckets.

8.2 Download a model from the model store

If you would rather download the model, and not load it into memory, you can use model store's `download()` function.

```
file_path = model_store.download(
   local_path=".", # Where to download the model to
   domain="example-model", # The model's domain
   model_id="model-id"  # Optional; the ID of the specific model
)
```

CHAPTER

NINE

RETRIEVING MODEL AND DOMAIN INFORMATION

This library enables you to query your model registry programmatically.

The examples below assume you have created a model store instance already:

```python
from modelstore.model_store import ModelStore

model_store = ModelStore.from_aws_s3(bucket_name)
```

9.1 Model domains

Models are uploaded into domains: a domain is created when you upload your first model to it. You can list all of the existing domains and get information about a specific domain with:

```python
model_domains = model_store.list_domains()

meta_data = model_store.get_domain("my-domain")
```

9.2 Model states

Model states are tags that can be used to control the lifecycle of models in a domain. To see the list of model states that have been created, use:

```python
model_states = model_store.list_model_states()
```

Note: there are some reserved states that modelstore uses to, for example, keep track of model IDs that have been deleted.

9.3 Model versions

Models are uploaded into domains: a domain is created when you upload your first model to it. You can list all of the existing domains and get information about a specific domain with:

```python
# List all models
model_ids = model_store.list_versions("my-domain")

```

(continues on next page)

(continued from previous page)

```
# List models with a given state
prod_model_ids = model_store.list_versions("my-domain", state_name="production")
```

9.4 Models

The main thing you can do with a model is download or load it back. You can also retrieve information about a specific model, and delete models from the registry.

```
# Get information about a specific model
meta_data = model_store.get_model_info("my-domain", "my-model")
```

CHAPTER TEN

DELETING MODELS

Deleting a model removes the files from the registry. If you query for a model that has been deleted, a `ModelDeletedException` is raised.

```
# Delete a model
model_store.delete_model("my-domain", "my-model", skip_prompt=True)

# Will raise a ModelDeletedException
meta_data = model_store.get_model_info("my-domain", "my-model")
```

CHAPTER

ELEVEN

CONTROLLING MODEL STATES

This library enables you to control models by setting their state. For example, you may want to set a model to have state “production.” You can then query the model store for models by state, and change model states.

The examples below assume you have created a model store instance already:

```
from modelstore.model_store import ModelStore

model_store = ModelStore.from_aws_s3(bucket_name)
```

11.1 Create a state

Before doing anything with a model state, you need to create it. This is a one-time operation.

```
production_state = "production"

model_store.create_model_state(production_state)
```

11.2 Set and unset a model’s state

Once a state has been created, you can add a model to a state. You can add a model to more than one state, and you can add more than one model to a state.

```
model_domain = "my-domain"
model_id = "my-model-id"
production_state = "production"

model_store.set_model_state(model_domain, model_id, state_name)
```

To unset a model’s state, you can use:

```
model_store.remove_model_state(model_domain, model_id, state_name)
```

11.3 Find models by state

After setting the state of one or more models, you can find them by adding the state name to the list versions function:

```
model_ids = modelstore.list_versions(
    model_domain,
    state_name=production_state
)
```

CHAPTER

TWELVE

MODELSTORE CLI COMMANDS

You can use modelstore (version > 0.0.71) from the command line to upload and download models:

```
# To upload a model
python -m modelstore upload <domain> </path/to/file>

# To download a model
python -m modelstore download <domain> <model-id>
```

Modelstore figures out how to read from your storage by looking for specific environment variables.

Your environment needs to define (1) a value for MODEL_STORE_STORAGE which tells modelstore what type of storage you are using, and (2) values that depend on the specific type of storage that you are using.

All of these are summarised in the table below:

Table 1: Storage environment variables

Storage	MODEL_STORE_STORAGE	Other environment variables
AWS s3	aws-s3	MODEL_STORE_AWS_BUCKET AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY
Azure Container	azure-container	MODEL_STORE_AZURE_CONTAINER AZURE_ACCOUNT_NAME AZURE_ACCESS_KEY AZURE_STORAGE_CONNECTION_STRING
Google Cloud Storage	google-cloud-storage	MODEL_STORE_GCP_PROJECT MODEL_STORE_GCP_BUCKET
File system	filesystem	MODEL_STORE_ROOT

CHAPTER

THIRTEEN

TROUBLESHOOTING

13.1 Common errors when setting up s3 storage

This page describes the steps you need to take to store models in s3.

Before you start, you will need to **create the s3 bucket you want to use**. The modelstore library does not create s3 buckets and assumes they exist already. To do this, you can follow the creating a bucket AWS documentation.

Next, install modelstore and boto3 in your Python environment:

```
pip install modelstore boto3
```

If you have not done this before, you will need to set up the AWS authentication credentials by following the boto3 configuration guide.

And you can then create a model store instance and point it to your bucket:

```python
from modelstore import ModelStore

model_store = ModelStore.from_aws_s3("my-bucket")
```

The remainder of this page describes some common errors you may run into. If you need further support, please create an issue on Github.

13.1.1 ModuleNotFoundError: boto3 is not installed

The model store library works with several different types of storage, and therefore does not install all of their libraries. If you see a `ModuleNotFoundError`, then you need to install boto3.

```
pip install boto3
```

The current version of modelstore requires `boto3>=1.6.16,<1.8`.

13.1.2 botocore.exceptions.NoCredentialsError: Unable to locate credentials

You will need to set up the AWS authentication credentials. As this documentation page describes, boto3 "looks at various configuration locations until it finds configuration values."

To start, follow the AWS documentation to get your access key and secret access key values. There are then two approaches you can use here.

Option 1: run `aws configure` by following the boto3 configuration guide.

```
 aws configure
AWS Access Key ID [None]: my-access-key
AWS Secret Access Key [None]: my-secret-access-key
```

Option 2: set environment variables.

```
export AWS_ACCESS_KEY_ID="my-access-key"
export AWS_SECRET_ACCESS_KEY="my-secret-access-key"
```

13.1.3 botocore.exceptions.ParamValidationError: Parameter validation failed

You'll see this error if you are passing a `bucket_name` that boto3 cannot parse. Note: you do not need to include the "s3://" in the bucket name.

```
>>> model_store = ModelStore.from_aws_s3("s3://my-bucket-name")
[...]
botocore.exceptions.ParamValidationError: Parameter validation failed:
Invalid bucket name "s3://my-bucket-name": Bucket name must match the regex "^[a-zA-Z0-9.
↪\-_]{1,255}$" or be an ARN matching the regex "^arn:(aws).*:(s3|s3-object-lambda):[a-z\
↪-0-9]*:[0-9]{12}:accesspoint[/:][a-zA-Z0-9\-.]{1,63}$|^arn:(aws).*:s3-outposts:[a-z\-0-
↪9]+:[0-9]{12}:outpost[/:][a-zA-Z0-9\-]{1,63}[/:]accesspoint[/:][a-zA-Z0-9\-]{1,63}$"
```

13.1.4 Exception: Failed to set up the AWSStorage storage

This exception is raised if modelstore can't read from the bucket you are pointing it to. With logging enabled, you will see this line when you try to create a model store instance:

```
>>> model_store = ModelStore.from_aws_s3("my-bucket-name")
Unable to access bucket: <bucket-name>

[...]
Exception: Failed to set up the AWSStorage storage
```

To resolve this, you can check:

1. Does the bucket exist? If not, you can follow the creating a bucket AWS documentation.
2. Is there a typo in the `bucket_name` variable?

13.1.5 botocore.exceptions.EndpointConnectionError: Could not connect to the endpoint URL

This exception is raised if modelstore can't connect to the s3 bucket. One way this happens is if you specify a region that is not a known value. The full list of regions is available on this AWS documentation page.

For example, if you use a region name, you'll see an error:

```
>>> model_store = ModelStore.from_aws_s3(bucket_name=os.environ["AWS_BUCKET_NAME"],␣
↪region="Frankfurt")
>>> model_store.list_domains()
[...]
raise EndpointConnectionError(endpoint_url=request.url, error=e)
botocore.exceptions.EndpointConnectionError: Could not connect to the endpoint URL:
↪"https://operator-ai-modelstore-direct.s3.Frankfurt.amazonaws.com/?list-type=2&
↪prefix=operatorai-model-store%2Fdomains&encoding-type=url"
```

But if you use the region code, it should not error:

```
>>> model_store = ModelStore.from_aws_s3(bucket_name=os.environ["AWS_BUCKET_NAME"],␣
↪region="eu-central-1")
>>> model_store.list_domains()
['diabetes-boosting-demo']
```

13.1.6 Seeing another exception?

If you need further support, please create an issue on Github.

This documentation is open source. If you would like to add anything to it, please open a pull request on Github.

This documentation is open source. If you would like to add anything to it, please open a pull request on Github.

CHAPTER

FOURTEEN

LICENSE

Copyright 2022 Neal Lathia

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER FIFTEEN

CONTACT

If you have any questions or feedback, feel free to open an issue on Github or email me: `neal.lathia@gmail.com` or