Machine Learning

Experiments

Date published: 2020-07-16
Date modified: 2024-03-05

https://docs.cloudera.com/

Legal Notice

Contents

Experiments with MLflow

Machine Learning requires experimenting with a wide range of datasets, data preparation steps, and algorithms to build a model that maximizes a target metric. Once you have built a model, you also need to deploy it to a production system, monitor its performance, and continuously retrain it on new data and compare it with alternative models.

CML lets you train, reuse, and deploy models with any library, and package them into reproducible artifacts that other data scientists can use.

CML packages the ML models in a reusable, reproducible form so you can share it with other data scientists or transfer it to production.

CML is compatible with the MLflow™ tracking API and makes use of the MLflow client library as the default method to log experiments. Existing projects with existing experiments are still available and usable.

The functionality described in this document is for the new version of the Experiments feature, which replaces an older version of the Experiments feature that could not be used from within Sessions. In Projects that have existing Experiments created using the previous feature, you can continue to view these existing Experiments. New projects use the new Experiments feature.

CML Experiment Tracking through MLflow API

CML's experiment tracking features allow you to use the MLflow client library for logging parameters, code versions, metrics, and output files when running your machine learning code. The MLflow library is available in CML Sessions without you having to install it. CML also provides a UI for later visualizing the results. MLflow tracking lets you log and query experiments using the following logging functions:

Note: CML currently supports only Python for experiment tracking.

- mlflow.create_experiment() creates a new experiment and returns its ID. Runs can be launched under the experiment by passing the experiment ID to mlflow.start_run.

  Cloudera recommends that you create an experiment to organize your runs. You can also create experiments using the UI.
- mlflow.set_experiment() sets an experiment as active. If the experiment does not exist, mlflow.set_experiment creates a new experiment. If you do not wish to use the set_experiment method, a default experiment is selected.

  Cloudera recommends that you set the experiment using mlflow.set_experiment.
- mlflow.start_run() returns the currently active run (if one exists), or starts a new run and returns a mlflow.Activ eRun object usable as a context manager for the current run. You do not need to call start_run explicitly; calling one of the logging functions with no active run automatically starts a new one.
- mlflow.end_run() ends the currently active run, if any, taking an optional run status.
- mlflow.active_run() returns a mlflow.entities.Run object corresponding to the currently active run, if any.

  Note: You cannot access currently-active run attributes (parameters, metrics, etc.) through the run returned by mlflow.active_run. In order to access such attributes, use the mlflow.tracking.MlflowClient as follows:

  ```
  client = mlflow.tracking.MlflowClient()
  data = client.get_run(mlflow.active_run().info.run_id).data
  ```
- mlflow.log_param() logs a single key-value parameter in the currently active run. The key and value are both strings. Use mlflow.log_params() to log multiple parameters at once.

- mlflow.log_metric() logs a single key-value metric for the current run. The value must always be a number. MLflow remembers the history of values for each metric. Use mlflow.log_metrics() to log multiple metrics at once.

  Parameters:

  - key - Metric name (string)
  - value - Metric value (float). Note that some special values such as +/- Infinity may be replaced by other values depending on the store. For example, the SQLAlchemy store replaces +/- Infinity with max / min float values.
  - step - Metric step (int). Defaults to zero if unspecified.

  Syntax - mlflow.log_metrics(metrics: Dict[str, float], step: Optional[int] = None) # None
- mlflow.set_tag() sets a single key-value tag in the currently active run. The key and value are both strings. Use mlflow.set_tags() to set multiple tags at once.
- mlflow.log_artifact() logs a local file or directory as an artifact, optionally taking an artifact_path to place it within the run's artifact URI. Run artifacts can be organized into directories, so you can place the artifact in a directory this way.
- mlflow.log_artifacts() logs all the files in a given directory as artifacts, again taking an optional artifact_path.
- mlflow.get_artifact_uri() returns the URI that artifacts from the current run should be logged to.

For more information on MLflow API commands used for tracking, see MLflow Tracking.

Running an Experiment using MLflow

This topic walks you through a simple example to help you get started with Experiments in Cloudera Machine Learning.

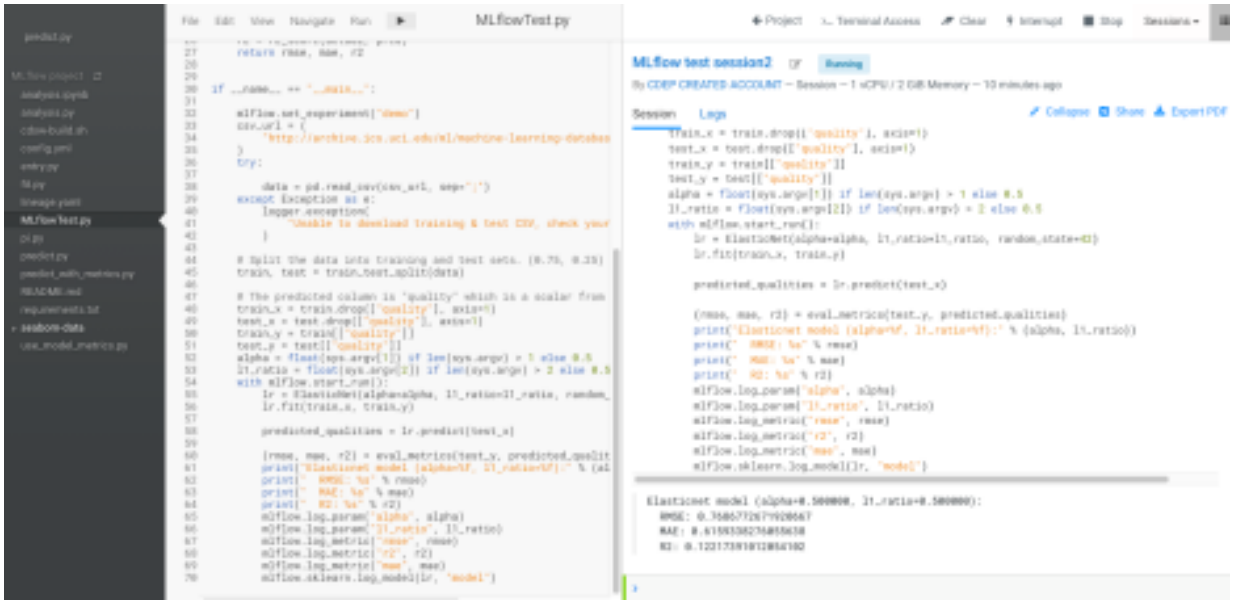
Best practice: It's useful to display two windows while creating runs for your experiments: one window displays the Experiments tab and another displays the MLflow Session.

1. From your Project window, click New Experiment and create a new experiment. Keep this window open to return to after you run your new session.
2. From your Project window, click New Session.
3. Create a new session using ML Runtimes. Experiment runs cannot be created from sessions using Legacy Engine.
4. In your Session window, import MLflow by running the following code: import mlflow The ML Flow client library is installed by default, but you must import it for each session.
5. Start a run and then specify the MLflow parameters, metrics, models and artifacts to be logged. You can enter the code in the command prompt or create a project. See *CML Experiment Tracking through MLflow API* for a list of functions you can use.

   For example:

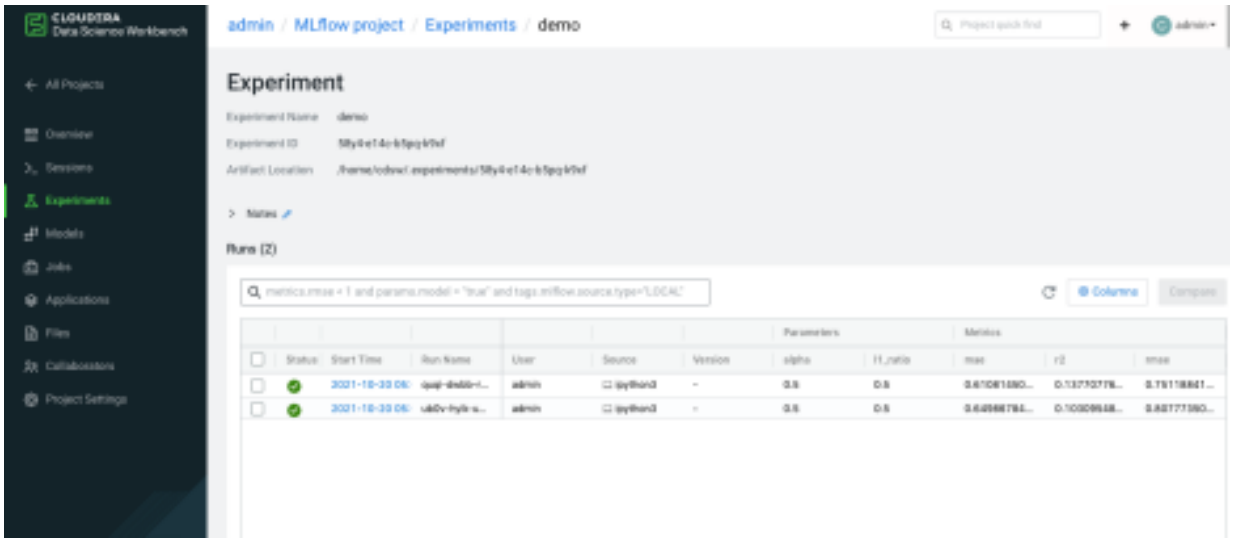
```
mlflow.set_experiment(<experiment_name>)
mlflow.start_run()
mlflow.log_param("input", 5)
mlflow.log_metric("score", 100)
with open("data/features.txt", 'w') as f:
    f.write(features)
# Writes all files in "data" to root artifact_uri/states
mlflow.log_artifacts("data", artifact_path="states")
## Artifacts are stored in project directory under
/home/cdsw/.experiments/<experiment_id>/<run_id>/artifacts
```

```
mlflow.end_run()<
```



For information on using editors, see Using Editors for ML Runtimes.

6. Continue creating runs and tracking parameters, metrics, models, and artifacts as needed.
7. To view your run information, display the Experiments window and select your experiment name. CML displays the Runs table.



8. Click the Refresh button on the Experiments window to display recently created runs
9. You can customize the Run table by clicking Columns, and selecting the columns you want to display.

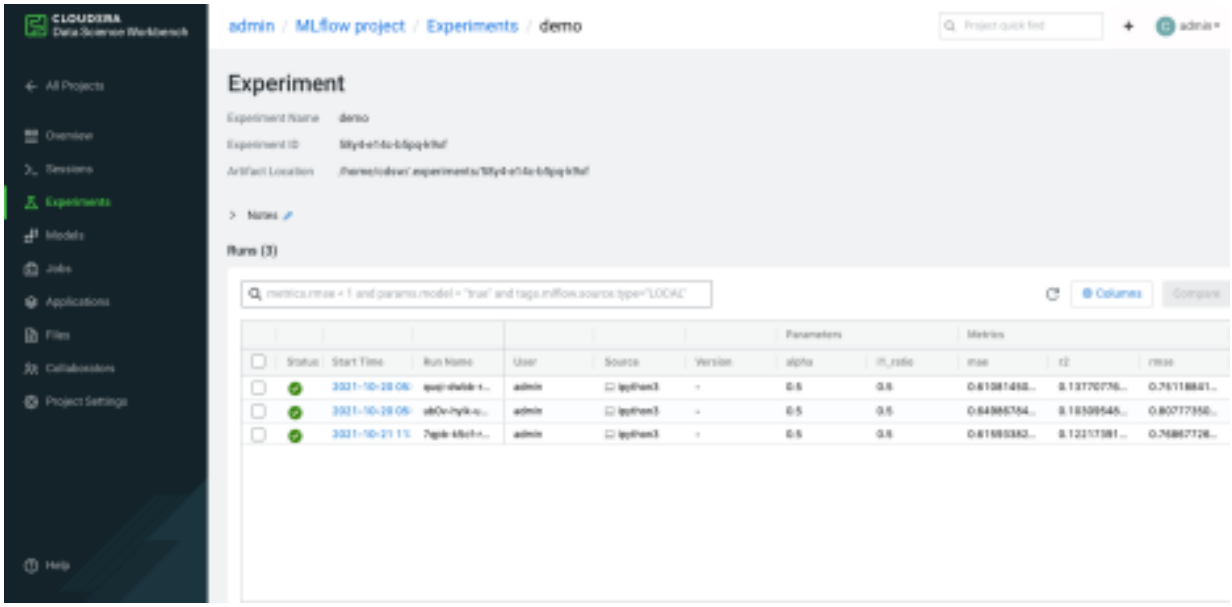
Related Information

Using Editors for ML Runtimes

Visualizing Experiment Results

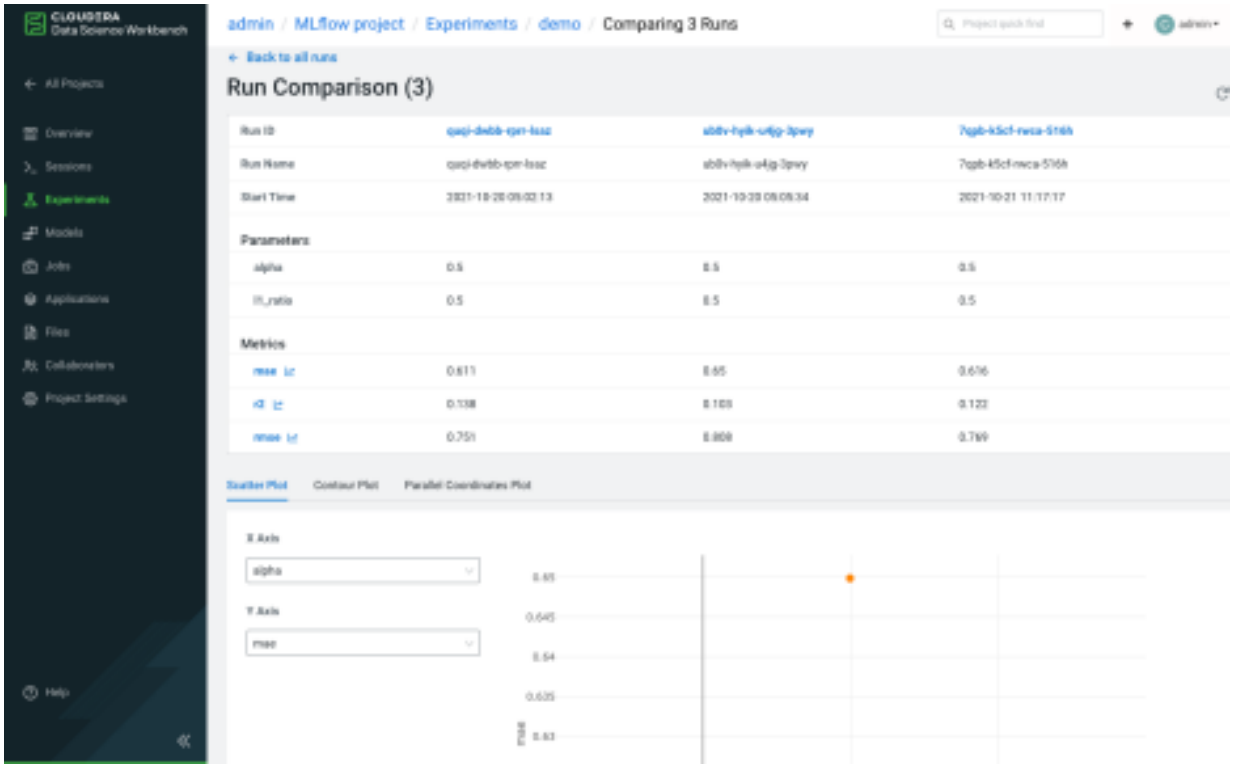
After you create multiple runs, you can compare your results.

1. Go to Experiments and click on your experiment name. CML displays the Runs table populated by all of the runs for the experiment.



2. You can search your run information by using the search field at the top of the Run table.
3. You can customize the Run table by clicking Columns, and selecting the columns you want to display.
4. You can display details for a specific run by clicking the start time for the run in the Run table. You can add notes for the run by clicking the Notes icon. You can display the run metrics in a chart format by clicking the specific metric under Metrics.
5. To compare the data from multiple runs, use the checkbox in the Run table to select the runs you want to compare. You can use the top checkbox to select all runs in the table. Alternatively, you can select runs using the spacebar and arrow keys.

6. Click Compare. Alternatively, you can press Cmd/Ctrl + Enter. CML displays a separate window containing a table titled Run Comparison and options for comparing your parameters and metrics.



This Run Comparison table lists all of the parameters and the most recent metric information from the runs you selected. Parameters that have changed are highlighted

7. You can graphically display the Run metric data by clicking the metric names in the Metrics section. If you have a single value for your metrics, it will display as a bar chart. If your run has multiple values, the metrics comparison page displays the information with multiple steps, for example, over time. You can choose how the data is displayed:
   - Time (Relative): graphs the time relative to the first metric logged, for each run.
   - Time (Wall): graphs the absolute time each metric was logged.
   - Step: graphs the values based on the cardinal order.
8. Below the Run Comparison table, you can choose how the Run information is displayed:
   - Scatter Plot: Use the scatter plot to see patterns, outliers, and anomalies.
   - Contour Plot: Contour plots can only be rendered when comparing a group of runs with three or more unique metrics or parameters. Log more metrics or parameters to your runs to visualize them using the contour plot.
   - Parallel Coordinates Plot: Choose the parameters and metrics you want displayed in the plot.

Using an MLflow Model Artifact in a Model REST API

You can use MLflow to create, deploy, and manage models as REST APIs to serve predictions

1. To create an MLflow model add the following information when you run an experiment:

```
mlflow.log_artifacts ("output")
```

```
mlflow.sklearn.log_model(lr, "model")
```

For example:

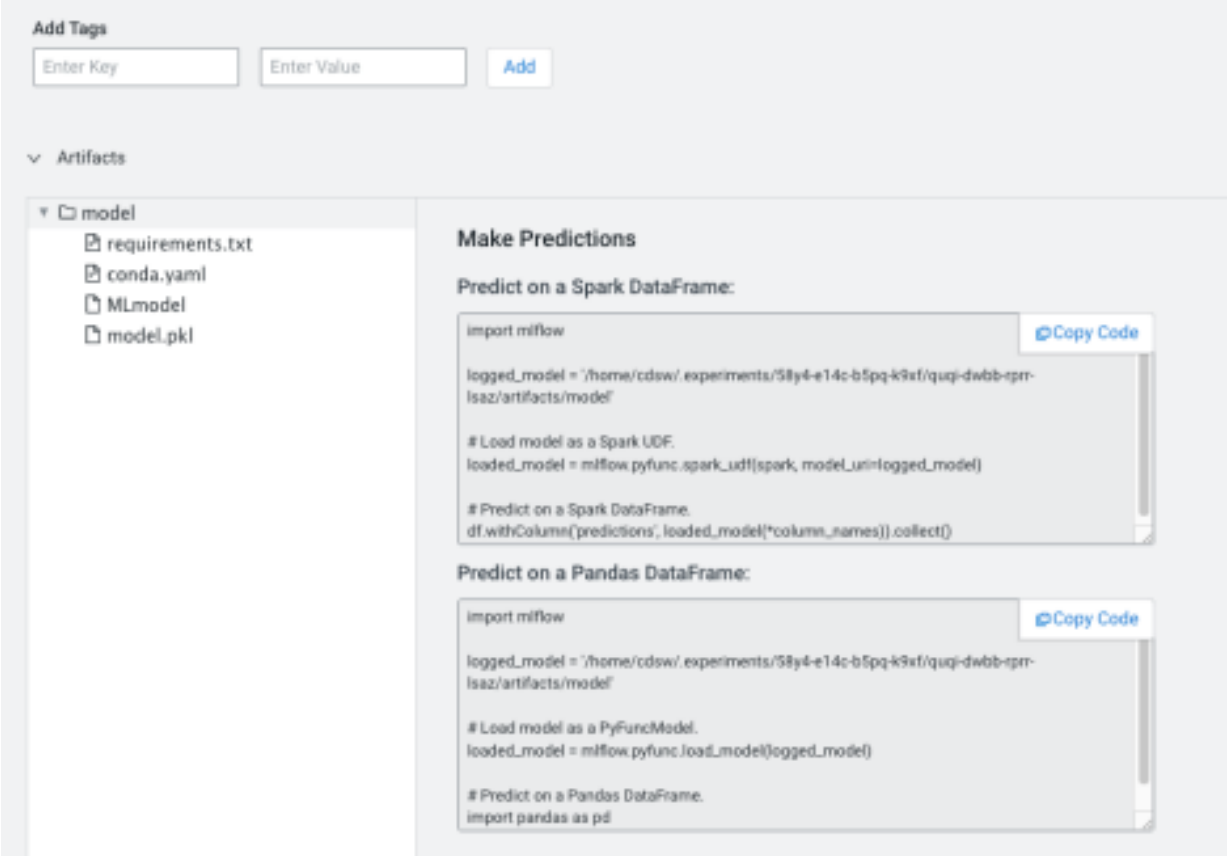
```
# The data set used in this example is from http://archive.ics.uci.edu/ml/
datasets/Wine+Quality
# P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis.
# Modeling wine preferences by data mining from physicochemical properti
es. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.
import os
import warnings
import sys

import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_sc
ore
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ElasticNet
from urllib.parse import urlparse
import mlflow
from mlflow.models import infer_signature
import mlflow.sklearn

import logging

logging.basicConfig(level=logging.WARN)
logger = logging.getLogger(__name__)

def eval_metrics(actual, pred):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mae = mean_absolute_error(actual, pred)
    r2 = r2_score(actual, pred)
    return rmse, mae, r2

if __name__ == "__main__":
    warnings.filterwarnings("ignore")
    np.random.seed(40)

    # Read the wine-quality csv file from the URL
    csv_url = (
        "https://raw.githubusercontent.com/mlflow/mlflow/master/tests/da
tasets/winequality-red.csv"
    )
    try:
        data = pd.read_csv(csv_url, sep=";")
    except Exception as e:
        logger.exception(
            "Unable to download training & test CSV, check your internet
 connection. Error: %s", e
        )

    # Split the data into training and test sets. (0.75, 0.25) split.
    train, test = train_test_split(data)

    # The predicted column is "quality" which is a scalar from [3, 9]
    train_x = train.drop(["quality"], axis=1)
    test_x = test.drop(["quality"], axis=1)
    train_y = train[["quality"]]
    test_y = test[["quality"]]

    alpha = float(sys.argv[1]) if len(sys.argv) > 1 else 0.5
    l1_ratio = float(sys.argv[2]) if len(sys.argv) > 2 else 0.5
```

```
    with mlflow.start_run():
        lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
        lr.fit(train_x, train_y)

        predicted_qualities = lr.predict(test_x)

        (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)
        print("Elasticnet model (alpha={:f}, l1_ratio={:f}):".format(al
pha, l1_ratio))
        print("  RMSE: %s" % rmse)
        print("  MAE: %s" % mae)
        print("  R2: %s" % r2)

        mlflow.log_param("alpha", alpha)
        mlflow.log_param("l1_ratio", l1_ratio)
        mlflow.log_metric("rmse", rmse)
        mlflow.log_metric("r2", r2)
        mlflow.log_metric("mae", mae)

        predictions = lr.predict(train_x)
        signature = infer_signature(train_x, predictions)
        mlflow.sklearn.log_model(lr, "model", signature=signature, regist
ered_model_name="testmodel")
```

In this example we are training a machine learning model using linear regression to predict wine quality. This script creates the MLflow model artifact and logs it to the model directory: /home/cdsw/.experiments/<experiment_id>/<run_id>/artifacts/models

2. To view the model, navigate to the Experiments page and select your experiment name. CML displays the Runs page and lists all of your current runs.
3. Click the run from step 1 that created the MLflow model. CML displays the Runs detail page

4. Click Artifacts to display a list of all the logged artifacts for the run.



5. Click model. CML displays the MLflow information you use to create predictions for your experiment.

Deploying an MLflow model as a CML Model REST API

In the future, you will be able to register models to a Model Registry and then deploy Model REST APIs with those models. Today, these models can be deployed using the following manual process instead

1. Navigate to your project. Note that models are always created within the context of a project.
2. Click Open Workbench and launch a new Python 3 session.
3. Create a new file within the project if one does not already exist: cdsw-build.sh This file defines the function that will be called when the model is run and will contain the MLflow prediction information.
4. Add the following information to the cdsw-build.sh file: pip3 install sklearn mlflow pandas
5. For non-Python template projects and old projects check the following.

   a. Check to make sure you have a .gitignore file. If you do not have the file, add it.
   b. Add the following information to the .gitignore file: !.experiments

   For new projects using a Python template, this is already present.
6. Create a Python file to call your model artifact using a Python function. For example:

   - Filename: mlpredict.py
   - Function: predict
7. Copy the MLflow model file path from the Make Predictions pane in the Artifacts section of the Experiments/Run details page and load it in the Python file. This creates a Python function which accepts a dictionary of the input variables and converts these to a Pandas data frame, and returns the model prediction. For example:

```
import mlflow
```

```
import pandas as pd
logged_model =
     '/home/cdsw/.experiments/7qwz-l620-d7v6-1922/glma-oqxb-szc7-c8hf/a
rtifacts/model'
def predict(args):
     # Load model as a PyFuncModel.
     data = args.get('input')
     loaded_model = mlflow.pyfunc.load_model(logged_model)
     # Predict on a Pandas DataFrame.
     return loaded_model.predict(pd.DataFrame(data))
```

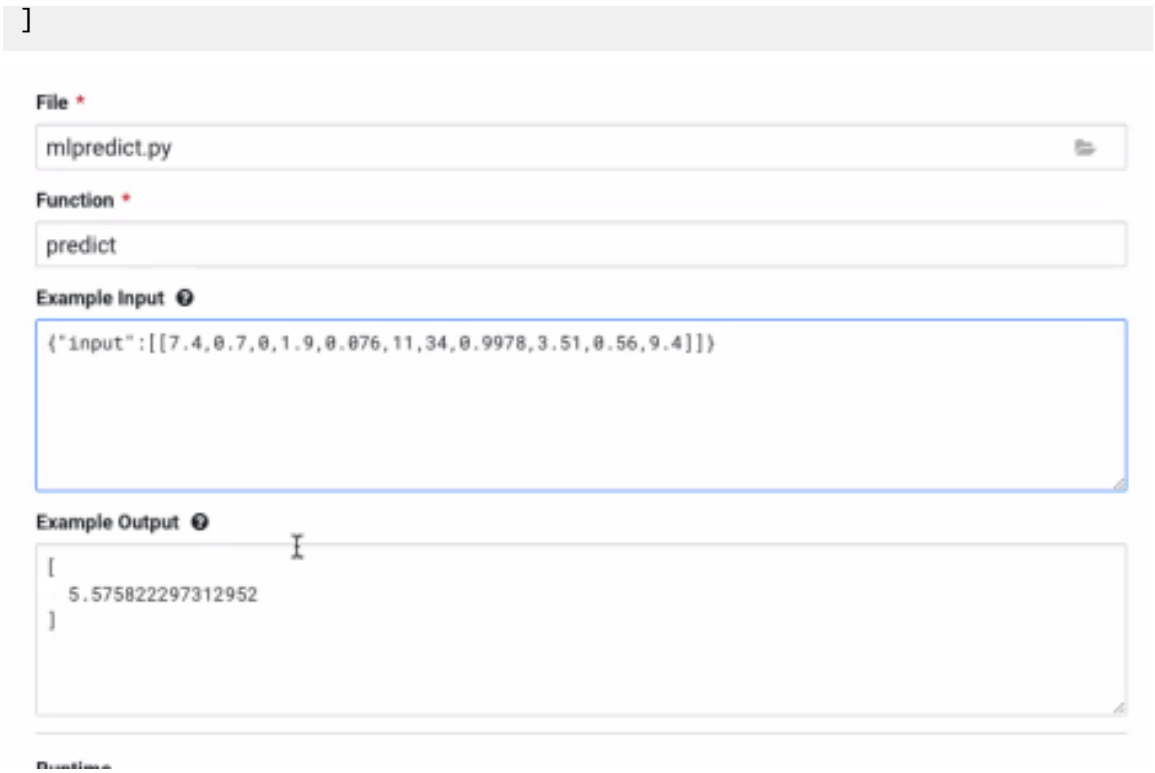
Note: In practice, do not assume that users calling the model will provide input in the correct format or enter good values. Always perform input validation.

8. Deploy the predict function to a REST endpoint.
   a. Go to the project Overview page
   b. Click Models New Model .
   c. Give the model a Name and Description
   d. Enter details about the model that you want to build. In this case:
      - File: mlpredict.py
      - Function: predict
      - Example Input:

```
{
"input": [
[7.4, 0.7, 0, 1.9, 0.076, 11, 34, 0.9978,
3.51, 0.56, 9.4]
]
}
```

      - Example output:

```
[
5.575822297312952
```



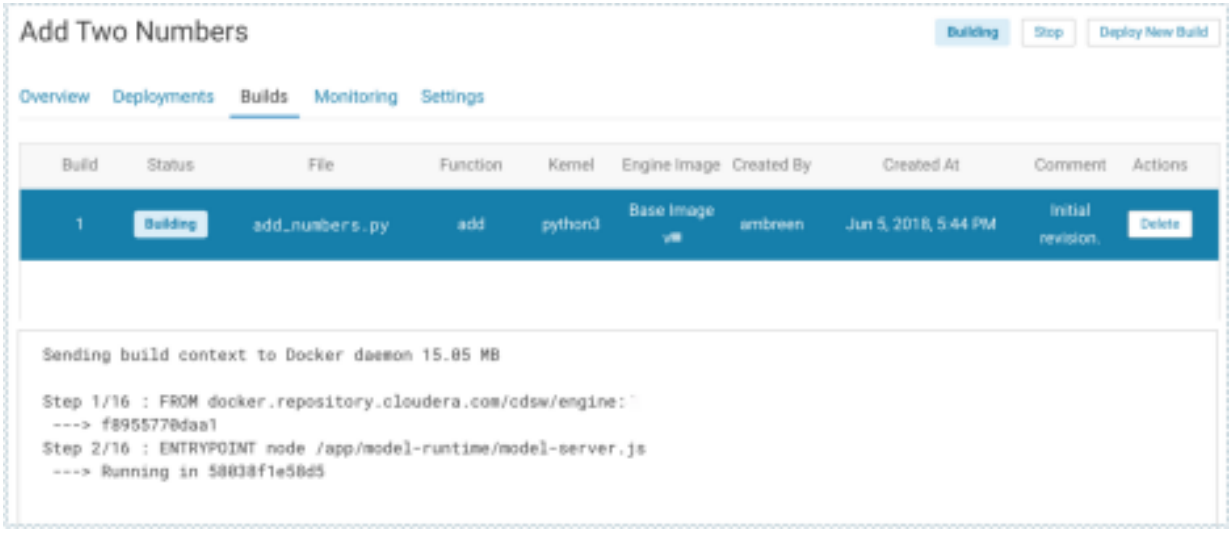
e. Select the resources needed to run this model, including any replicas for load balancing.

Note: The list of options here is specific to the default engine you have specified in your Project Settings: ML Runtimes or Legacy Engines. Engines allow kernel selection, while ML Runtimes allow Editor, Kernel, Variant, and Version selection. Resource Profile list is applicable for both ML Runtimes and Legacy Engines.

f. Click Deploy Model.

9. Click on the model to go to its Overview page.

10. Click Builds to track realtime progress as the model is built and deployed. This process essentially creates a Docker container where the model will live and serve requests.



11. Once the model has been deployed, go back to the model Overview page and use the Test Model widget to make sure the model works as expected. If you entered example input when creating the model, the Input field will be pre-populated with those values.

12. Click Test. The result returned includes the output response from the model, as well as the ID of the replica that served the request.

    Model response times depend largely on your model code. That is, how long it takes the model function to perform the computation needed to return a prediction. It is worth noting that model replicas can only process one request at a time. Concurrent requests will be queued until the model can process them.

1.0 and 2.0 scoring API differences

The format for calling the model has changed between MLflow versions 1.0 and 2.0.

The MLflow Model scoring protocol has changed in MLflow version 2.0. If you are seeing an error, you are likely using an outdated scoring request format. To resolve the error, either update your request format or adjust your MLflow Model's requirements file to specify an older version of MLflow (for example, change the 'mlflow' requirement specifier to mlflow==1.30.0). If you are making a request using the MLflow client (e.g. via mlflow.pyfunc.spark_udf()), upgrade your MLflow client to a version >= 2.0 in order to use the new request format.

For more information about the updated MLflow Model scoring protocol in MLflow 2.0, see *MLflow Models*.

Related Information

MLflow Models

Automatic Logging

Automatic logging allows you to log metrics, parameters, and models without the need for an explicit log statement.

You can perform autologging two ways:

1. Call mlflow.autolog() before your training code. This will enable autologging for each supported library you have installed as soon as you import it.
2. Use library-specific autolog calls for each library you use in your code. See below for examples.

For more information about the libraries supported by autologging, see Automatic Logging.

Setting Permissions for an Experiment

Experiments are associated with the project ID, so permissions are inherited from the project. If you want to allow a colleague to view the experiments of a project, you should give them Viewer (or higher) access to the project.

Known issues and limitations

CML has the following known issues and limitations with experiments and MLflow.

- CML currently supports only Python for experiment tracking.
- Experiment runs cannot be created from MLFlow on sessions using Legacy Engine. Instead, create a session using an ML Runtime.
- The version column in the runs table is empty for every run. In a future release, this will show a git commit sha for projects using git.
- There is currently no mechanism for registering a model to a Model Registry. In a future release, you will be able to register models to a Model Registry and then deploy Model REST APIs with those models.
- Browsing an empty experiment will display a spinner that doesn’t go away.
- Running an experiment from the workbench (from the dropdown menu) refers to legacy experiments and should not be used going forward.

- Tag/Metrics/Parameter columns that were previously hidden on the runs table will be remembered, but CML won't remember hiding any of the other columns (date, version, user, etc.)
- Admins can not browse all experiments. They can only see their experiments on the global Experiment page.
- Performance issues may arise when browsing the run details of a run with a lot of metric results, or when comparing a lot of runs.
- Runs can not be deleted or archived.

MLflow transformers

This is an example of how MLflow transformers can be supported in Cloudera Machine Learning.

Note: This is an experimental feature.

This example shows how to implement a translation workflow using a translation model.

1. Save the following as a file, for example, named mlflowtest.py.

```
#!pip3 install torch transformers torchvision tensorflow

    import mlflow
    from mlflow.models import infer_signature
    from mlflow.transformers import generate_signature_output
    from transformers import pipeline

    en_to_de = pipeline("translation_en_to_de")

    data = "MLflow is great!"
    output = generate_signature_output(en_to_de, data)
    #signature = infer_signature(data, output)

    with mlflow.start_run() as run:
    mlflow.transformers.log_model(
    transformers_model=en_to_de,
    artifact_path="english_to_german_translator",
    input_example=data,
    registered_model_name="entodetranslator",
    )


    model_uri = f"runs:/{run.info.run_id}/english_to_german_translator"
    loaded = mlflow.pyfunc.load_model(model_uri)

    print(loaded.predict(data))
```

2. In the Model Registry page, find the entodetranslator model. Deploy the model.
3. Make a request using the following payload:

```
{
  "dataframe_split": {
    "columns": [
      "data"
    ],
    "data": [
      [
        "MLflow is great!"
      ]
    ]
  }
```

```
}
```

4. In a session, run the mlflowtest.py file. It should print the following output.

```
print(loaded.predict(data))
['MLflow ist großartig!']
```

Note: For more information, see mlflow.transformers.