MLflow: Understanding Runs & Experiments

MLflow structures its tracking mechanism around two core concepts: **Experiments** and **Runs**. Below is a detailed breakdown of these concepts:

Experiments

In MLflow, an **Experiment** is a logical grouping of related runs. It can be thought of in multiple ways:

1. **As a "Project"**: All runs related to a particular project are grouped. For instance, all model versions and attempts related to "Predicting House Prices" can be grouped under one experiment.
2. **As an "Approach"**: Runs can be grouped based on a specific methodology. For instance, grouping runs related to "Deep Learning Models for House Price Prediction" separately from "Tree-Based Models for House Price Prediction".

Runs

A **Run** in MLflow represents a single execution of your machine learning or data science code. Details captured in a run include:

- **Source of Execution**: Includes specific commit hash for reproducibility and details about the executed code segment.
- **Artifacts**: Output files generated during the run, like trained model files or plots.
- **Parameters**: Input configurations such as hyperparameters.
- **Metrics**: Performance indicators like accuracy, RMSE, F1-score, etc.

Example:

Consider a project focused on predicting customer churn.

Experiment 1: "Customer Churn Prediction - Logistic Regression"

- **Run 1**:
  - **Source**: Commit hash `a1b2c3d4`
  - **Artifacts**: `logistic_model_v1.pkl`
  - **Parameters**: `learning_rate=0.01, max_iter=100`
  - **Metrics**: `accuracy=0.85, F1=0.8`
- **Run 2**:
  - **Source**: Commit hash `e5f6g7h8`
  - **Artifacts**: `logistic_model_v2.pkl`
  - **Parameters**: `learning_rate=0.05, max_iter=200`
  - **Metrics**: `accuracy=0.88, F1=0.82`

Experiment 2: "Customer Churn Prediction - Random Forest"

- **Run 1**:
  - **Source**: Commit hash `i9j0k1l2`

- **Artifacts**: `rf_model_v1.pkl`
- **Parameters**: `n_estimators=100, max_depth=10`
- **Metrics**: `accuracy=0.89, F1=0.84`

This structured approach of MLflow ensures systematic and organized tracking of all machine learning endeavors.

MLflow: Storage Mechanisms for Experiments and Runs

MLflow offers a versatile system for tracking and recording machine learning experiments and runs. The location and method of this storage can be configured based on the user's requirements.

Storage Options in MLflow

1. **Local Files (`mlruns` Directory)**:
   - By default, when MLflow is used without specifying a tracking URI, it records data locally within the `mlruns` directory.
   - This directory is generated in the current working directory of the executed script.
   - Within `mlruns`, data is systematically organized: experiments are stored in individual subdirectories, and each of these contains separate directories for every run. These run directories capture metrics, parameters, and artifacts.
2. **Remote Tracking Server**:
   - MLflow can be set up with a centralized server for tracking, ideal for collaboration among multiple users.
   - This server can be located on-premises or in the cloud, allowing centralized access to MLflow data.
3. **Relational Databases (via SQLAlchemy)**:
   - MLflow can utilize SQLAlchemy, a SQL toolkit and ORM for Python, to store its data in relational databases.
   - This setup is useful for larger teams or projects where data needs to be accessed and managed in a structured manner.

Local Setup: The `mlruns` Directory

When starting with MLflow, the local setup using the `mlruns` directory is often the most straightforward approach:

- Upon logging any data (metrics, parameters, artifacts) without specifying a tracking URI, MLflow will either create or append data to the `mlruns` directory.

Example:

Suppose you're experimenting with a simple linear regression model. The following Python code demonstrates this:

```
import mlflow
from sklearn.metrics import mean_squared_error

# Your data preprocessing and model training code here...

predictions = model.predict(X_test)
mse = mean_squared_error(y_test, predictions)

mlflow.log_metric("MSE", mse)
```

Once this script is run, given that no external tracking URI is provided, MLflow will store data in the `mlruns` directory.

For a visual representation and further interaction with the stored data, the MLflow UI can be launched:

```
mlflow ui
```

Executing this command will activate a local server, and you can then access the MLflow interface in a web browser. This interface allows you to inspect, compare, and delve deeper into your experiments and runs.

MLflow Storage and Deployment Scenarios

MLflow offers a spectrum of deployment and storage options, catering to different needs, from individual experimentation to complex enterprise-level operations.

Scenario 2: MLflow on localhost with SQLite

When MLflow is configured with SQLite, it employs a local SQLite database for data storage, diverging from the default file-based method.

How it Works:

- SQLite, known for its lightweight nature, is a file-based database system.
- All MLflow data, encompassing metrics, parameters, etc., gets stored in a structured manner inside an SQLite database file, typically ending with a `.db` extension.

Example:

```
import mlflow

mlflow.set_tracking_uri('sqlite:///mlflow.db')
```

This command initializes an SQLite database named mlflow.db where all MLflow data is stored.

Scenario 3: MLflow on localhost with Tracking Server

In this configuration, MLflow utilizes a standalone tracking server set up on the user's local machine. This provides a more centralized interface for tracking experiments and runs.

How it Works:

- The tracking server acts as a centralized hub for managing MLflow data, while still being hosted on the user's local system.
- It's an excellent method for users looking to familiarize themselves with the MLflow tracking server functionalities without committing to a remote deployment.

Usage Example:

To initiate the tracking server on your local machine:

```
mlflow server
```

And in your MLflow-enabled Python script or project:

```
import mlflow

# Pointing to the local tracking server
mlflow.set_tracking_uri('http://localhost:5000')
```

This will ensure that any logs, metrics, or artifacts from your project are sent to the local tracking server you've started.

To start the MLflow tracking server on a different port, such as 5001, you can use the `--port argument` when launching the server. Here's how you can do it:

```
mlflow server --port 5001
```

After starting the server on port 5001, you would then point your MLflow client to this port. In your Python code, you'd set the tracking URI as follows:

```
import mlflow

# Pointing to the local tracking server on port 5001
mlflow.set_tracking_uri('http://localhost:5001')
```

Exmple - Tracking a Linear regression model

Training the model:

First, train a linear regression model that takes two hyperparameters: alpha and l1_ratio

```
# 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
properties. In Decision Support Systems, Elsevier, 47(4):547-553, 2009.

import logging
import sys
import warnings
from urllib.parse import urlparse

import numpy as np
import pandas as pd
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
from sklearn.model_selection import train_test_split

import mlflow
import mlflow.sklearn
from mlflow.models import infer_signature

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/datasets/win
equality-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(f"Elasticnet model (alpha={alpha:f}, l1_ratio=
{l1_ratio:f}):")
        print(f"  RMSE: {rmse}")
        print(f"  MAE: {mae}")
        print(f"  R2: {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)

        tracking_url_type_store =
urlparse(mlflow.get_tracking_uri()).scheme

        # Model registry does not work with file store
        if tracking_url_type_store != "file":
            # Register the model
            # There are other ways to use the Model Registry, which
depends on the use case,
            # please refer to the doc for more information:
            # https://mlflow.org/docs/latest/model-registry.html#api-
workflow
            mlflow.sklearn.log_model(
                lr, "model", registered_model_name="ElasticnetWineModel",
signature=signature
```

```
            )
        else:
            mlflow.sklearn.log_model(lr, "model", signature=signature)
```

MLflow explanations (above code)

- mlflow.start_run():

This initiates a new MLflow run to log information about this training session. A run represents a single execution of your code, and each run can record parameters, metrics, tags, and artifacts. Everything you log via MLflow during the execution of your code inside this context will be associated with this run.

- Logging Parameters:

```
mlflow.log_param("l1_ratio", l1_ratio)
```

These lines log hyperparameters of your model. Parameters are the configurable parts of your training run and typically are set before the start of a training session. They can be things like learning rates, batch sizes, or any other argument to your script.

- Logging Metrics:

```
mlflow.log_metric("rmse", rmse)
mlflow.log_metric("r2", r2)
mlflow.log_metric("mae", mae)
```

Metrics are numerical values that you want to optimize. They are the measurable outputs of your experiment, like accuracy or loss values. In this case, you're recording the Root Mean Squared Error (RMSE), R-squared (R2), and Mean Absolute Error (MAE) of your model.

- `infer_signature`

```
predictions = lr.predict(train_x)
signature = infer_signature(train_x, predictions)
```

infer_signature is a utility function from MLflow that automatically generates the signature of the model, which describes the schema of the model's inputs and outputs. The signature defines input and output column names and data types. This can be useful when deploying the model as it provides a safety check that input data matches the model's expectations. Let's dive more in `infer_signature` function of mlflow tracking package.

What is `infer_signature` ?

When deploying models to production, especially in a team or for external consumers, it's useful to have a clear contract or schema that describes what kind of data the model expects for input and what it will produce as output. This schema or contract is referred to as the "signature" of the model.

MLflow's `infer_signature` is a utility function that can automatically determine the input and output schema (or signature) of your model. It does this by inspecting sample input and output data you provide.

Why use it?

The signature provides a safety mechanism when deploying models. By defining the expected input and output schema, you can catch inconsistencies or errors in data formatting before they cause larger issues. When a model is served in production with a defined signature, input data can be checked against the signature before passing it to the model, ensuring data integrity and consistency.

Example:

Let's say you have a simple linear regression model that predicts house prices based on two features: number of bedrooms and square footage. Here's how you might use

`infer_signature`:

```
import mlflow
from mlflow.models.signature import infer_signature
from sklearn.linear_model import LinearRegression
import pandas as pd

# Sample data
data = {
    'bedrooms': [2, 3, 3, 4, 4],
    'square_feet': [1500, 1600, 1700, 1800, 1900],
    'price': [200000, 220000, 240000, 280000, 300000]
}

df = pd.DataFrame(data)

# Train a model
X = df[['bedrooms', 'square_feet']]
y = df['price']

model = LinearRegression()
model.fit(X, y)

# Predict
predictions = model.predict(X)

# Infer the signature
signature = infer_signature(X, predictions)

print(signature)
```

The printed signature might look something like:

```
signature:
inputs: '[Column(name=bedrooms, type=double), Column(name=square_feet,
type=double)]'
outputs: 'Column(name=price, type=double)'
```

- Logging the Model

```
if tracking_url_type_store != "file":
    mlflow.sklearn.log_model(
        lr, "model", registered_model_name="ElasticnetWineModel",
signature=signature
    )
else:
    mlflow.sklearn.log_model(lr, "model", signature=signature)
```

This section checks the type of tracking store (whether it's a local file system or a remote tracking server). If it's not a local file store, the model is logged and registered under the name "ElasticnetWineModel". If it is a local file store, only the model is logged without registering.

- mlflow.sklearn.log_model: This logs the trained model to the tracking server. Logging the model means that MLflow records both the model itself and its metadata in a format that makes it easy to later deploy as a web service, among other things.
- registered_model_name: When provided, MLflow will also register this model under the given name in the Model Registry, which provides collaboration and versioning of models.

he mlflow.sklearn.log_model function is a key utility in MLflow's toolkit, especially when working with Scikit-learn models. Let's dissect it:

- Function overview:

mlflow.sklearn.log_model() is used to log a Scikit-learn model into MLflow's tracking system. By "logging," we mean that it stores the model in a format that MLflow understands and can later use for tasks like deployment or further analysis. Parameters of this function are as follow:

1. Model (`lr` in the code): This is the trained Scikit-learn model that you want to log
2. Artifact Path (`model` in the code): This is the path within the MLflow run's artifact URI where the model will be saved. Think of it as a sub-directory within your run's storage location. Typically, this is just set to "model", but you can provide a different name or nested paths if you prefer a different organizational structure.
3. `signature`: This is the model's signature, which describes the schema of the model's inputs and outputs. As discussed in the previous section, this signature provides a clear contract of what the model expects as input and what it will produce as output.

> Example: Imagine you're training a simple logistic regression model on the famous Iris dataset:

```
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from mlflow.models.signature import infer_signature

# Load and split data
data = load_iris(as_frame=True)
X_train, X_test, y_train, y_test = train_test_split(data.data,
data.target, test_size=0.2, random_state=42)

# Train a model
clf = LogisticRegression(max_iter=200)
clf.fit(X_train, y_train)

# Infer the signature
input_example = X_train.head(1)
signature = infer_signature(input_example, clf.predict(input_example))

# Log the model to MLflow
with mlflow.start_run():
    mlflow.sklearn.log_model(clf, "iris_model", signature=signature)
```

Accessing the MLflow UI:

Once you've logged some runs using MLflow as in the wine example, you can use the MLflow UI to visualize and compare them.

- Start the MLflow UI: If you're using the default local file store (i.e., you didn't set a tracking URI and the runs are saved to an mlruns directory in your project), navigate to your project directory and run:

```
mlflow ui
```

If you want to use a specific port (e.g., 5001), you can specify it with:

```
mlflow ui --port 5001
```

Access via a Web Browser: Open a web browser and go to:

```
http://localhost:5000
```

Adding more models to the `./mlruns` directory:

- Linear regression:

```
from sklearn.linear_model import LinearRegression

with mlflow.start_run(run_name="Linear Regression"):
    # Train the model
    lr_model = LinearRegression()
    lr_model.fit(train_x, train_y)

    # Predict
    lr_predictions = lr_model.predict(test_x)

    # Evaluate
    (lr_rmse, lr_mae, lr_r2) = eval_metrics(test_y, lr_predictions)

    # Log model and metrics
    mlflow.log_param("Model", "Linear Regression")
    mlflow.log_metric("rmse", lr_rmse)
    mlflow.log_metric("r2", lr_r2)
    mlflow.log_metric("mae", lr_mae)

    mlflow.sklearn.log_model(lr_model, "linear_regression_model")
```

- ElasticNet Regression: We already have one variant, but let's try a couple more combinations:

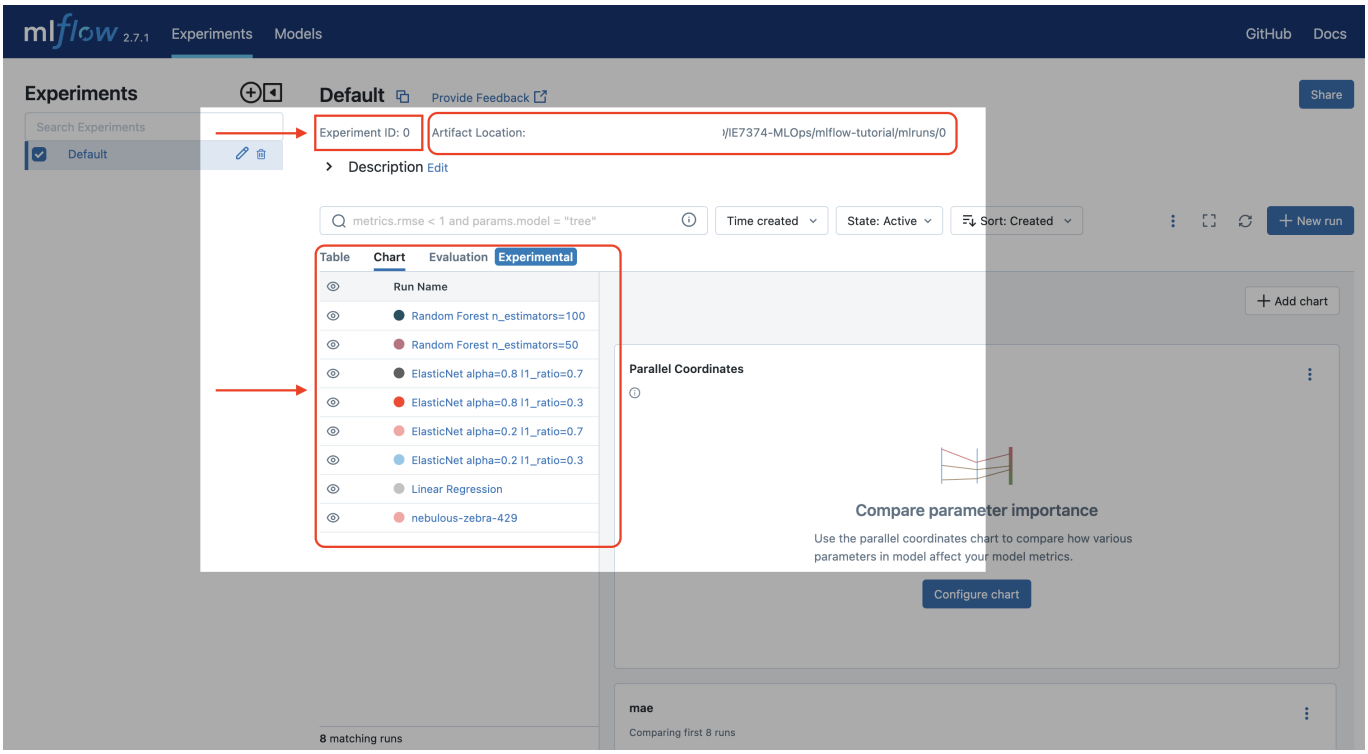
```
alphas = [0.2, 0.8]
l1_ratios = [0.3, 0.7]

for alpha in alphas:
    for l1_ratio in l1_ratios:
        with mlflow.start_run(run_name=f"ElasticNet alpha={alpha}
l1_ratio={l1_ratio}"):
            # Train the model
            en_model = ElasticNet(alpha=alpha, l1_ratio=l1_ratio,
random_state=42)
            en_model.fit(train_x, train_y)

            # Predict
            en_predictions = en_model.predict(test_x)

            # Evaluate
            (en_rmse, en_mae, en_r2) = eval_metrics(test_y,
en_predictions)

            # Log model and metrics
            mlflow.log_param("Model", "ElasticNet")
            mlflow.log_param("alpha", alpha)
            mlflow.log_param("l1_ratio", l1_ratio)
            mlflow.log_metric("rmse", en_rmse)
            mlflow.log_metric("r2", en_r2)
```

```
            mlflow.log_metric("mae", en_mae)

            mlflow.sklearn.log_model(en_model,
f"elasticnet_alpha_{alpha}_l1_{l1_ratio}")
```

- Random Forest Regressor:

```
from sklearn.ensemble import RandomForestRegressor

n_estimators_values = [50, 100]

for n_estimators in n_estimators_values:
    with mlflow.start_run(run_name=f"Random Forest n_estimators=
{n_estimators}"):
        # Train the model
        rf_model = RandomForestRegressor(n_estimators=n_estimators,
random_state=42)
        rf_model.fit(train_x, train_y.values.ravel())

        # Predict
        rf_predictions = rf_model.predict(test_x)

        # Evaluate
        (rf_rmse, rf_mae, rf_r2) = eval_metrics(test_y, rf_predictions)

        # Log model and metrics
        mlflow.log_param("Model", "Random Forest")
        mlflow.log_param("n_estimators", n_estimators)
        mlflow.log_metric("rmse", rf_rmse)
        mlflow.log_metric("r2", rf_r2)
        mlflow.log_metric("mae", rf_mae)

        mlflow.sklearn.log_model(rf_model,
f"random_forest_n_{n_estimators}")
```

Now let's run `!mlflow ui --port=5001` in the notebook and see how mlflow UI looks like:



- `run_name` in `mlflow.start_run()`: The run_name parameter in mlflow.start_run() specifies the name of that specific run. In the context of MLflow:
- A Run is an individual execution of your data science code. Each run can record parameters, metrics, tags, and artifacts. For example, if you're trying different hyperparameters for a model, the `run_name` might be something like `Random Forest n_estimators=100`, making it clear that this run used a Random Forest model with 100 estimators.

Changing name of an experiment to the current date:

```
import mlflow
from mlflow.tracking import MlflowClient
from datetime import date

# Set the new experiment name to the current date
new_experiment_name = str(date.today())

# Create an instance of MlflowClient
client = MlflowClient()

# Rename the experiment
client.rename_experiment("0", new_experiment_name)
```

Loading one of the runs:

we use `mlflow.pyfunc.load_model(model_path)` to load one of the runs. For doing so, we need the run ID which we can find in artifacts section of our run in mlfow UI. For instance, we are predicting on one sampele of wine dataset using one of the runs which is random forest regressor:

```
import mlflow.pyfunc

# Define the path to the model
model_path =
'mlruns/0/4bfbba9a0b194be7a2036197e88b054a/artifacts/random_forest_n_100'

# Load the model
loaded_model = mlflow.pyfunc.load_model(model_path)

sample_input = {
    "fixed acidity": [8.5],
    "volatile acidity": [0.28],
    "citric acid": [0.56],
    "residual sugar": [1.1],
    "chlorides": [0.092],
    "free sulfur dioxide": [17.0],
    "total sulfur dioxide": [60.0],
    "density": [0.9954],
    "pH": [3.30],
    "sulphates": [0.75],
    "alcohol": [10.5]
}

# Convert to DataFrame for prediction
import pandas as pd
sample_df = pd.DataFrame(sample_input)

# Predict
prediction = loaded_model.predict(sample_df)
print(prediction)
```

Installing dependencies:

> For installing all the dependencies of one the runs tracked by mlflow tracking, in each articats of each run there is a `requirements.txt` file that we can install all required libraries using below command:

```
!pip install -r
mlruns/0/4bfbba9a0b194be7a2036197e88b054a/artifacts/random_forest_n_100/re
quirements.txt
```

Serving the Model

- Sering the model:

First, decide which model you want to serve. For this example, let's assume you want to serve the model stored at this path: `mlruns/0/aea73ab36d544af29927cf26199b8888/artifacts/model`

we can run the following command in a terminal (ensure you're in the directory where your mlruns folder is located and your virtual environment is activated):

```
MODEL_PATH="mlruns/0/aea73ab36d544af29927cf26199b8888/artifacts/model"
mlflow models serve -m $MODEL_PATH -h 0.0.0.0 -p 5001
```

This will start a local server on port 5001. Wait until you see a message indicating the server has started.

- Send Data for Predictions: Next, we'll need a sample input from the wine dataset to send for predictions. Let's use Python to do this:

```
sample_data = {
    "fixed acidity": 8.5,
    "volatile acidity": 0.28,
    "citric acid": 0.56,
    "residual sugar": 1.1,
    "chlorides": 0.092,
    "free sulfur dioxide": 17.0,
    "total sulfur dioxide": 60.0,
    "density": 0.9954,
    "pH": 3.30,
    "sulphates": 0.75,
    "alcohol": 10.5
}

import requests

# Define the URL of the deployed model
url = 'http://localhost:5001/invocations'

# Sample data for prediction
data = {
    "columns": list(sample_data.keys()),
    "instances": [list(sample_data.values())]
}
# print(data)

# Send the request
response = requests.post(url, json=data)

# Extract and print predictions
predictions = response.json()
print(predictions)
```

Output should look like below:

```
{'predictions': [6.15]}
```