MLflow and Transformers

Welcome to our tutorial on leveraging the power of **Transformers** with **MLflow**. This guide is designed for beginners, focusing on machine learning workflows and model management.

What Will You Learn?

In this lab, you will learn how to:

- Set up a simple text generation pipeline using the Transformers library.
- Log the model and its parameters using MLflow.
- Infer the input and output signature of the model automatically.
- Simulate serving the model using MLflow and make predictions with it.

  Introduction to Transformers

  Transformers are a type of deep learning model that have revolutionized natural language processing (NLP). Developed by 🤗 Hugging Face, the Transformers library offers a variety of state-of-the-art pre-trained models for NLP tasks.

Why Combine MLflow with Transformers?

Integrating MLflow with Transformers offers numerous benefits:

- **Experiment Tracking**: Log and compare model parameters and metrics.
- **Model Management**: Track different model versions and their performance.
- **Reproducibility**: Document all aspects needed to reproduce predictions.
- **Deployment**: Simplify deploying Transformers models to production.

Now, let's dive into the world of MLflow and Transformers!

```
# Disable tokenizers warnings when constructing pipelines
%env TOKENIZERS_PARALLELISM=false

import warnings

# Disable a few less-than-useful UserWarnings from setuptools and pydantic
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
```

Imports and Pipeline configuration

In this first section, we are setting up our environment and configuring aspects of the transformers pipeline that we'll be using to generate a text response from the LLM.

Setting up our Pipeline

- **Import**: We import the necessary libraries: transformers for building our NLP model and mlflow for model tracking and management.
- **Task Definition**: We then define the task for our pipeline, which in this case is `text2text-generation`` This task involves generating new text based on the input text.
- **Pipeline Declaration**: Next, we create a generation_pipeline using the `pipeline` function from the Transformers library. This pipeline is configured to use the `declare-lab/flan-alpaca-large model`, which is a pre-trained model suitable for text generation.
- **Input Example**: For the purposes of generating a signature later on, as well as having a visual indicator of the expected input data to our model when loading as a `pyfunc`, we next set up an input_example that contains sample prompts.

- **Inference Parameters**: Finally, we define parameters that will be used to control the behavior of the model during inference, such as the maximum length of the generated text and whether to sample multiple times.

Understanding Pipelines

Pipelines are a high-level abstraction provided by the Transformers library that simplifies the process of using models for inference. They encapsulate the complexity of the underlying code, offering a straightforward API for a variety of tasks, such as text classification, question answering, and in our case, text generation.

The `pipeline()` function

The pipeline() function is a versatile tool that can be used to create a pipeline for any supported task. When you specify a task, the function returns a pipeline object tailored for that task, constructing the required calls to sub-components (a tokenizer, encoder, generative model, etc.) in the order needed to fulfill the needs of the specified task. This abstraction dramatically simplifies the code required to use these models and their respective components.

Task-Specific Pipelines

In addition to the general pipeline() function, there are task-specific pipelines for different domains like audio, computer vision, and natural language processing. These specialized pipelines are optimized for their respective tasks and can provide additional convenience and functionality.

Benefits of Using Pipelines

Using pipelines has several advantages:

- **Simplicity**: You can perform complex tasks with a minimal amount of code.
- **Flexibility**: You can specify different models and configurations to customize the pipeline for your needs.
- **Efficiency**: Pipelines handle batching and dataset iteration internally, which can lead to performance improvements.

Due to the utility and simple, high-level API, MLflow's `transformers` implementation uses the `pipeline` abstraction by default (although it can support component-only mode as well).

```
import transformers

import mlflow

# Define the task that we want to use (required for proper pipeline construction)
task = "text2text-generation"

# Define the pipeline, using the task and a model instance that is applicable for our
task.
generation_pipeline = transformers.pipeline(
    task=task,
    model="declare-lab/flan-alpaca-large",
)

# Define a simple input example that will be recorded with the model in MLflow, giving
# users of the model an indication of the expected input format.
input_example = ["prompt 1", "prompt 2", "prompt 3"]

# Define the parameters (and their defaults) for optional overrides at inference time.
parameters = {"max_length": 512, "do_sample": True, "temperature": 0.4}
```

Introduction to Model Signatures in MLflow

In the realm of machine learning, model signatures play a crucial role in ensuring that models receive and produce the expected data types and structures. MLflow includes a feature for defining model signatures, helping to standardize and enforce correct model usage.

Quick Learning Points

- **Model Signature Purpose**: Ensures consistent data types and structures for model inputs and outputs.
- **Visibility and Validation**: Visible in MLflow's UI and validated by MLflow's deployment tools.
- **Signature Types**: Column-based for tabular data, tensor-based for tensor inputs/outputs, and with params for models requiring additional inference parameters.

Understanding Model Signatures

A model signature in MLflow describes the schema for inputs, outputs, and parameters of an ML model. It is a blueprint that details the expected data types and shapes, facilitating a clear interface for model usage. Signatures are particularly useful as they are:

- Displayed in MLflow's UI for easy reference.
- Employed by MLflow's deployment tools to validate inputs during inference.
- Stored in a standardized JSON format alongside the model's metadata.

The Role of Signatures in Code

In the following section, we are using MLflow to infer the signature of a machine learning model. This involves specifying an input example, generating a model output example, and defining any additional inference parameters. The resulting signature is used to validate future inputs and document the expected data formats.

Types of Model Signatures

Model signatures can be:

- **Column-based**: Suitable for models that operate on tabular data, with each column having a specified data type and optional name.
- **Tensor-based**: Designed for models that take tensors as inputs and outputs, with each tensor having a specified data type, shape, and optional name.
- **With Params**: Some models require additional parameters for inference, which can also be included in the signature.

For the transformers flavor, all input types are of the Column-based type (referred to within MLflow as `ColSpec` types).

Signature Enforcement

MLflow enforces the signature at the time of model inference, ensuring that the provided input and parameters match the expected schema. If there's a mismatch, MLflow will raise an exception or issue a warning, depending on the nature of the mismatch.

```
# Generate the signature for the model that will be used for inference validation and
type checking (as well as validation of parameters being submitted during inference)
signature = mlflow.models.infer_signature(
    input_example,
    mlflow.transformers.generate_signature_output(generation_pipeline, input_example),
    parameters,
)

# Visualize the signature
signature
```

Creating an experiment

We create a new MLflow Experiment so that the run we're going to log our model to does not log to the default experiment and instead has its own contextually relevant entry.

```
mlflow.set_tracking_uri("http://127.0.0.1:8081")

mlflow.set_experiment("Transformers Introduction")
```

Logging the Transformers Model with MLflow

We log our model with MLflow to manage its lifecycle efficiently and keep track of its versions and configurations.

Logging a model in MLflow is a crucial step in the model lifecycle management, enabling efficient tracking, versioning, and management. The process involves registering the model along with its essential metadata within the MLflow tracking system.

Utilizing the mlflow.transformers.log_model function, specifically tailored for models and components from the `transformers` library, simplifies this task. This function is adept at handling various aspects of the models from this library, including their complex pipelines and configurations.

When logging the model, crucial metadata such as the model's signature, which was previously established, is included. This metadata plays a significant role in the subsequent stages of the model's lifecycle, from tracking its evolution to facilitating its deployment in different environments. The signature, in particular, ensures the model's compatibility and consistent performance across various platforms, thereby enhancing its utility and reliability in practical applications.

By logging our model in this way, we ensure that it is not only well-documented but also primed for future use, whether it be for further development, comparative analysis, or deployment.

A Tip for Saving Storage Cost

When you call mlflow.transformers.log_model, MLflow will saves a full copy of the Transformers model weight. However, this could take large storage space (3GB for `flan-alpaca-large` model), but might be redundant when you don't modify the model weight, because it is exactly same as the one you can download from the HuggingFace model hub.

To avoid the unnecessary copy, you can use 'reference-only' save mode which is introduced in MLflow 2.11.0, by setting `save_pretrained=False` when logging or saving the Transformer model. This tells MLflow not to save the copy of the base model weight, but just a reference to the HuggingFace Hub repository and version, hence more storage-efficient and faster in time. For more details about this feature, please refer to Storage-Efficient Model Logging.

```
with mlflow.start_run():
    model_info = mlflow.transformers.log_model(
        transformers_model=generation_pipeline,
        artifact_path="text_generator",
        input_example=input_example,
        signature=signature,
        # Uncomment the following line to save the model in 'reference-only' mode:
        # save_pretrained=False,
    )
```

Loading the Text Generation Model

We initialize our text generation model using MLflow's pyfunc module for seamless model loading and interaction.

The `pyfunc` module in MLflow serves as a generic wrapper for Python functions. Its application in MLflow facilitates the loading of machine learning models as standard Python functions. This approach is especially advantageous for models logged or registered via MLflow, streamlining the interaction with the model regardless of its training or serialization specifics.

Utilizing mlflow.pyfunc.load_model, our previously logged text generation model is loaded using its unique model URI. This URI is a reference to the stored model artifacts. MLflow efficiently handles the model's deserialization, along with any associated dependencies, preparing it for immediate use.

Once the model, referred to as `sentence_generator`, is loaded, it operates as a conventional Python function. This functionality allows for the generation of text based on given prompts. The model encompasses the complete process of inference, eliminating the need for manual input preprocessing or output postprocessing. This encapsulation not only simplifies model interaction but also ensures the model's adaptability for deployment across various platforms.

```
# Load our pipeline as a generic python function
sentence_generator = mlflow.pyfunc.load_model(model_info.model_uri)
```

Formatting Predictions for Tutorial Readability

Please note that the following function, `format_predictions`, is used only for enhancing the readability of model predictions within this Jupyter Notebook environment. It **is not a standard component** of the model's inference pipeline.

```
def format_predictions(predictions):
    """
    Function for formatting the output for readability in a Jupyter Notebook
    """
    formatted_predictions = []

    for prediction in predictions:
        # Split the output into sentences, ensuring we don't split on abbreviations or
initials
        sentences = [
            sentence.strip() + ("." if not sentence.endswith(".") else "")
            for sentence in prediction.split(". ")
            if sentence
        ]

        # Join the sentences with a newline character
        formatted_text = "\n".join(sentences)

        # Add the formatted text to the list
        formatted_predictions.append(formatted_text)

    return formatted_predictions
```

Generating Predictions with Custom Parameters

In this section, we demonstrate generating predictions using a sentence generator model with custom parameters. This includes prompts for selecting weekend activities and requesting a joke.

Quick Overview

- **Scenario**: Generating text for different prompts.
- **Custom Parameter**: Overriding the `temperature` parameter to control text randomness.
- **Default Values**: Other parameters use their defaults unless explicitly overridden.

  Prediction Process Explained

  We use the `predict` method on the `sentence_generator` pyfunc model with a list of string prompts, including:

- A request for help in choosing between hiking and kayaking for a weekend activity.
- A prompt asking for a joke related to hiking.

To influence the generation process, we override the `temperature` parameter. This parameter impacts the randomness of the generated text:

- **Lower Temperature**: Leads to more predictable and conservative outputs.
- **Higher Temperature**: Fosters varied and creative responses.

Utilizing Custom Parameters

In this example, the `temperature` parameter is explicitly set for the prediction call. Other parameters set during model logging will use their default values, unless also overridden in the `params` argument of the prediction call.

Output Formatting

The `predictions` variable captures the model's output for each input prompt. We can format these outputs for enhanced readability in the following steps, presenting the generated text in a clear and accessible manner.

```
# Validate that our loaded pipeline, as a generic pyfunc, can produce an output that
makes sense
predictions = sentence_generator.predict(
    data=[
        "I can't decide whether to go hiking or kayaking this weekend. Can you help me
decide?",
        "Please tell me a joke about hiking.",
    ],
    params={"temperature": 0.7},
)

# Format each prediction for notebook readability
formatted_predictions = format_predictions(predictions)

for i, formatted_text in enumerate(formatted_predictions):
    print(f"Response to prompt {i+1}:\n{formatted_text}\n")
```

Closing Remarks

This demonstration showcases the flexibility and power of the model in generating contextually relevant and creative text responses. By formatting the outputs, we ensure that the results are not only accurate but also presented in a manner that is easy to read and understand, enhancing the overall user experience within this Jupyter Notebook environment.