Certified AI Practitioner Week 04 Call 01 - Deploying Open Models the Right Way (From 🤗 to SageMaker)

Learning Objectives

- Understand what Hugging Face is and its role in modern GenAI workflows
- Decide when to use Hugging Face models vs Amazon Bedrock
- Deploy a Hugging Face model on SageMaker using the `HuggingFaceModel` class
- Invoke a real-time text classification endpoint and interpret the results
- Explain how SageMaker manages Hugging Face models using containers, IAM, and S3

🤗 What is Hugging Face?

Hugging Face is one of the most important open-source communities in machine learning today.

It provides:

- **The Hub** – A massive library of pre-trained models for tasks like sentiment analysis, translation, summarization, Q&A, image classification, audio transcription, and more.
- **Transformers library** – A Python library for using and fine-tuning state-of-the-art deep learning models from across research and industry.
- **Datasets and Tokenizers** – Tools to make preprocessing, tokenization, and evaluation easier across domains.
- **Community Contributions** – Most models on Hugging Face are shared by researchers, developers, and companies around the world.

You can use Hugging Face models:

- Locally (with Python)
- Through APIs (hosted by Hugging Face)
- Or in the cloud (via SageMaker, Vertex AI, etc.)

In this notebook, we'll use Hugging Face models **on SageMaker** using a managed container, without needing to build a custom Docker image.

🤗 Why Hugging Face?

Hugging Face is an open-source platform and community centered around modern machine learning — especially natural language processing (NLP).

It provides:

- A massive hub of pre-trained models covering text, vision, and audio tasks
- Tools like `transformers`, `datasets`, and `tokenizers` for building with ML
- A friendly ecosystem for researchers, engineers, and enterprises

With Hugging Face:

- ✅ You can browse and deploy thousands of open models
- ✅ Fine-tune models for your data
- ✅ Run locally or in cloud platforms like SageMaker

Popular Tasks:
Text Classification · Summarization · Question Answering · Translation · Embeddings · Image Recognition · Speech-to-Text

When to Use Hugging Face vs Bedrock

Use Case	Bedrock	Hugging Face
Managed security	✅ Yes	❌ You manage IAM + roles
Plug-and-play UX	✅ Playground, no code	❌ Requires code and SDK
Bring your own model	❌ Not yet supported	✅ Upload or fine-tune your own
Community models / open	❌ Limited to AWS partners	✅ Full access to public models

Use Case	Bedrock	Hugging Face
CI/CD and customization	🔶 Limited	✅ Full flexibility

Infrastructure Setup with CloudFormation

To train models in SageMaker Studio, we first need to provision the necessary infrastructure.

In this step, we'll use an automated **CloudFormation template** to create:

- A **SageMaker Studio domain** for running cloud-based notebooks
- An **IAM execution role** with S3 and SageMaker permissions
- A dedicated **S3 bucket** to store training data and model artifacts

Instead of clicking through the AWS Console, we'll deploy this setup programmatically using `boto3`. The stack will output everything you need - including the bucket name and role ARN - to use in the next steps of this notebook.

In [7]:

```
import boto3
import json

def stack_exists(name):
    try:
        cf.describe_stacks(StackName=name)
        return True
    except cf.exceptions.ClientError as e:
        if "does not exist" in str(e):
            return False
        raise  # re-raise any unexpected error

def deploy_stack(stack_name, template_body, parameters):
    if stack_exists(stack_name):
        print(f"🔄 Updating stack: {stack_name}")
        try:
            response = cf.update_stack(
                StackName=stack_name,
                TemplateBody=template_body,
                Parameters=parameters,
                Capabilities=["CAPABILITY_NAMED_IAM"]
```

```python
            )
            waiter = cf.get_waiter("stack_update_complete")
        except cf.exceptions.ClientError as e:
            if "No updates are to be performed" in str(e):
                print("✅ No changes detected.")
                # Print outputs
                outputs = cf.describe_stacks(StackName=stack_name)["Stacks"][0]["Outputs"]
                print("🔧 Stack Outputs:")
                print(json.dumps({o["OutputKey"]: o["OutputValue"] for o in outputs}, indent=2))
                return outputs
            else:
                raise
    else:
        print(f"🚀 Creating stack: {stack_name}")
        response = cf.create_stack(
            StackName=stack_name,
            TemplateBody=template_body,
            Parameters=parameters,
            Capabilities=["CAPABILITY_NAMED_IAM"]
        )
        waiter = cf.get_waiter("stack_create_complete")

    print(f"⏳ Waiting for {stack_name} to complete...")
    waiter.wait(StackName=stack_name)
    print("✅ Stack operation completed.")

    # Print outputs
    outputs = cf.describe_stacks(StackName=stack_name)["Stacks"][0]["Outputs"]
    print("🔧 Stack Outputs:")
    print(json.dumps({o["OutputKey"]: o["OutputValue"] for o in outputs}, indent=2))
    return outputs

ec2 = boto3.client("ec2")
cf = boto3.client("cloudformation")

# Get the default VPC ID
vpc_id = ec2.describe_vpcs(Filters=[{"Name": "isDefault", "Values": ["true"]}])["Vpcs"][0]["VpcId"]

# Get a public subnet ID in that VPC
subnets = ec2.describe_subnets(Filters=[{"Name": "vpc-id", "Values": [vpc_id]}])
subnet_id = subnets["Subnets"][0]["SubnetId"]
```

```python
# Load your template
with open("cf_templates/sagemaker_infra.yaml") as f:
    template_body = f.read()

bucketNameSuffix = "zali"
stack_name = "caip04-cloud-ml-stack"

parameters = [
    {"ParameterKey": "BucketNameSuffix", "ParameterValue": bucketNameSuffix},
    {"ParameterKey": "VpcId", "ParameterValue": vpc_id},
    {"ParameterKey": "SubnetId", "ParameterValue": subnet_id}
]

outputs = deploy_stack(stack_name, template_body, parameters)
```

```
🔄 Updating stack: caip04-cloud-ml-stack
⏳ Waiting for caip04-cloud-ml-stack to complete...
✅ Stack operation completed.
🔧 Stack Outputs:
{
  "StudioUserName": "caip04-user",
  "BucketName": "caip04-ml-bucket-zali",
  "DomainId": "d-p7phnt0viq28",
  "RoleArn": "arn:aws:iam::458806987020:role/caip04-execution-role-zali"
}
```

The Hugging Face Ecosystem: Models, Datasets, and Spaces

When you visit huggingface.co, you'll see three main categories at the top:

Models

This is the **model hub** — where you'll find thousands of pre-trained and fine-tuned machine learning models.

You can filter by:

- Task (e.g., sentiment analysis, summarization, translation)
- Framework (e.g., PyTorch, TensorFlow)

- Language or modality (text, image, audio)

In this notebook, we'll use one of these models:
`distilbert-base-uncased-finetuned-sst-2-english`

Datasets

The **Datasets** tab contains hundreds of benchmark and real-world datasets used for training, testing, and evaluation.

You'll find:

- Benchmark datasets like SST-2, SQuAD, IMDB
- Real-world data for tasks like classification, translation, QA, etc.
- Tools for loading and preprocessing directly with `datasets` library

---

Spaces

Spaces are **interactive AI web apps** — powered by models from the hub.

- Built using Gradio or Streamlit
- Visual interface for testing models without code
- Used by companies and individuals to showcase ML apps

✅ Spaces make models **interactive**
✅ Datasets make models **trainable**
✅ Models make ML **reusable**

What Is the Transformers Library?

The `transformers` library by Hugging Face is the most widely used Python package for working with **pretrained deep learning models** — especially **transformer-based architectures** like BERT, GPT, T5, RoBERTa, DistilBERT, and more.

It provides:

- ✅ Simple APIs for loading models, tokenizers, and pipelines
- ✅ Access to thousands of models on the Hugging Face Hub
- ✅ Compatibility with PyTorch, TensorFlow, and JAX
- ✅ Built-in support for tasks like:
    - Text classification
    - Summarization
    - Question answering
    - Translation
    - Text generation
    - Zero-shot classification
    - Token classification (NER)
    - Image and audio tasks (via multi-modal models)

Example: Sentiment Analysis in One Line

```python
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
result = classifier("I love this course!")
print(result)
```

> Output:

```
[{'label': 'POSITIVE', 'score': 0.999}]
```

Why It Matters

- You don't need to train or fine-tune anything to get started.
- Pretrained models + pipelines = instant value from research-grade ML.
- The same models used in academic papers, production systems, and Hugging Face Spaces are accessible in just a few lines of code.

For this lesson, we'll let SageMaker **host one of these models for us**, using the `transformers` library under the hood.

Example Model: distilbert-base-uncased-finetuned-sst-2-english

This is a lightweight, pre-trained **DistilBERT** model fine-tuned on the SST-2 dataset for **binary sentiment analysis**.

- **Task:** `text-classification`
- **Classes:** `POSITIVE` or `NEGATIVE`
- **Input:** A string of text (e.g., a product review or tweet)
- **Output:** A label with a confidence score

We'll deploy this model using the **Hugging Face prebuilt container** for SageMaker, which supports `transformers` -based models with no need to build custom Docker images.

About the Model Page on Hugging Face Hub

You can view this model live on the Hugging Face website:

distilbert-base-uncased-finetuned-sst-2-english

This page gives you everything you need to use the model:

What You'll Find on the Page

- **Model card**:
  Includes an overview of what the model does, the dataset it was trained on (SST-2), and the intended use.

- **Widget**:
  You can test the model directly in your browser by entering text and getting a sentiment result (`POSITIVE` or `NEGATIVE`).

- **Usage code**:
  Shows how to load and run the model using `transformers` in Python.

Example:

```
from transformers import pipeline
classifier = pipeline("sentiment-analysis", model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")
classifier("This is awesome!")
```

- **Tags**:
  Tells you this model supports tasks like `text-classification`, and that it runs on `pytorch`.

- **Model files**:
  Shows the underlying files (config, tokenizer, weights) that SageMaker will automatically pull when deploying.

Why This Page Matters

- Hugging Face model pages are the **source of truth** for model metadata.
- You'll reference these pages often when building GenAI pipelines in production.
- SageMaker's Hugging Face container can **deploy any model from the hub using just this model ID**.

Deploy the Hugging Face Model to SageMaker

Now that we've chosen our model (`distilbert-base-uncased-finetuned-sst-2-english`), we'll deploy it using the **SageMaker HuggingFaceModel class**, which makes it easy to host models from the 🤗 Hub without custom Docker images.

We specify:

- The model ID from Hugging Face
- The task (e.g., `text-classification`)
- The runtime environment (Transformers, PyTorch, Python)
- The IAM role to grant SageMaker permissions

Then we deploy the model to a managed SageMaker endpoint.

What Happens Behind the Scenes?

- SageMaker pulls a prebuilt container image for Hugging Face Transformers.
- It downloads the model weights and tokenizer from the 🤗 Hub.
- It spins up a new endpoint (`ml.t2.medium`) for real-time inference.

Notes

- This model performs binary sentiment analysis (`POSITIVE` or `NEGATIVE`).
- The endpoint will incur cost while it is running — delete it when you're done.

In [9]:
```
import boto3
import sagemaker
from sagemaker.huggingface import HuggingFaceModel

roleArn = {o["OutputKey"]: o["OutputValue"] for o in outputs if o["OutputKey"] == "RoleArn"}["RoleArn"]

# Define model configuration
hub = {
    'HF_MODEL_ID':'distilbert-base-uncased-finetuned-sst-2-english',
    'HF_TASK':'text-classification'
}

# Create HuggingFaceModel object
huggingface_model = HuggingFaceModel(
    transformers_version='4.26',
    pytorch_version='1.13',
    py_version='py39',
    env=hub,
    role=roleArn
)

# Deploy the model to an endpoint
predictor = huggingface_model.deploy(
    initial_instance_count=1,
    instance_type='ml.t2.medium',
    endpoint_name='huggingface-text-endpoint'
)
```

```
-------------!
```

Trigger the Deployed Endpoint via Boto3

Once your model is deployed to a SageMaker real-time endpoint, you can send it input using the `boto3` SageMaker Runtime client.

Function Overview

We define a reusable function: `trigger_huggingface_text_endpoint()`

- It takes in a **text prompt**
- Sends the prompt to the deployed endpoint using `invoke_endpoint()`
- Parses and returns the JSON prediction result

Input Format

The input must be structured as:

```
{ "inputs": "Your text here" }
```

In [ ]:

```python
import boto3
import json

endpoint_name = "huggingface-text-endpoint"

# Create the runtime client
runtime = boto3.client("sagemaker-runtime", region_name="us-east-1")

def trigger_huggingface_text_endpoint(runtime, prompt):

    # Prepare the input
    payload = {
        "inputs": prompt
    }
```

```
    # Call the endpoint
    response = runtime.invoke_endpoint(
        EndpointName=endpoint_name,
        ContentType="application/json",
        Body=json.dumps(payload)
    )

    # Parse and print the result
    result = json.loads(response['Body'].read())
    return result
```

Inference Examples: Sentiment in Action

After deploying the Hugging Face model and invoking it using our function, we tested several real-world phrases to observe how the model classifies sentiment.

Example 1: Positive Sentiment

In [19]:
```
result = trigger_huggingface_text_endpoint(runtime, "This class is amazing!")
result
```

Out[19]:
```
[{'label': 'POSITIVE', 'score': 0.9998821020126343}]
```

✅ The model confidently detects strong positive emotion.

Example 2: Negative Sentiment

In [17]:
```
result = trigger_huggingface_text_endpoint(runtime, "This class is horrible!")
result
```

Out[17]:
```
[{'label': 'NEGATIVE', 'score': 0.9997609257698059}]
```

✅ Correctly detects the strong negative tone with high confidence.

Example 3: Informal/Slang Language

In [18]:
```
result = trigger_huggingface_text_endpoint(runtime, "This class is the bomb!")
result
```

Out[18]:
```
[{'label': 'NEGATIVE', 'score': 0.9928885102272034}]
```

⚠️ **Why this is interesting**:
Although the phrase is slang for something very good, the model misclassifies it due to lack of context or training on informal expressions.

Teaching Point

Pretrained models work best with **standard, literal phrasing**. If your audience uses informal or domain-specific language, you may need to:

- Fine-tune the model
- Add post-processing logic
- Use few-shot prompting (if using a foundation model instead)

This is a great opportunity to talk about **limitations of out-of-the-box models** and the value of customization.

Where is it running?

- Model image is pulled from ECR
- Weights are downloaded from Hugging Face Hub
- Inference runs inside a SageMaker-managed container

Cleanup: Delete the Endpoint and Configuration

After testing your SageMaker-hosted model, it’s important to delete the endpoint and its configuration to avoid ongoing charges.

SageMaker real-time endpoints are **always-on** and billed by the hour, even if you’re not sending traffic.

What to Delete

1. **The Endpoint** – This is the live model you invoked.
2. **The Endpoint Configuration** – This defines the settings (instance type, model, etc.).

Both usually share the same name unless explicitly renamed.

In [ ]:
```
import boto3

client = boto3.client("sagemaker", region_name="us-east-1")
endpoint_name = "huggingface-text-endpoint"

# Delete the endpoint
try:
    client.delete_endpoint(EndpointName=endpoint_name)
    print(f"Deleted endpoint: {endpoint_name}")
except client.exceptions.ClientError as e:
    print(f"Could not delete endpoint: {e}")

# Delete the endpoint config (same name as endpoint if not renamed)
try:
    client.delete_endpoint_config(EndpointConfigName=endpoint_name)
    print(f"Deleted endpoint config: {endpoint_name}")
except client.exceptions.ClientError as e:
    print(f"Could not delete endpoint config: {e}")
```

Cleanup: Delete the CloudFormation Stack

If you created infrastructure (like IAM roles or S3 buckets) using a CloudFormation template, you should also clean it up once you're done.

The script below deletes your entire CloudFormation stack and waits for the deletion to complete.

What This Deletes

- IAM roles
- S3 buckets
- SageMaker-related permissions
- Any other resources created in the stack

Be careful — this will **permanently delete** all resources provisioned by the stack.

In [ ]:
```
import boto3

cf = boto3.client("cloudformation")
stack_name = "caip04-cloud-ml-stack"

def delete_stack_and_wait(stack_name):
    print(f"🧨 Deleting stack: {stack_name}")

    # Initiate deletion
    cf.delete_stack(StackName=stack_name)

    # Wait until stack deletion is complete
    waiter = cf.get_waiter("stack_delete_complete")
    print("⏳ Waiting for stack to be fully deleted...")
    waiter.wait(StackName=stack_name)

    print(f"✅ Stack '{stack_name}' successfully deleted.")

# Call it
delete_stack_and_wait(stack_name)
```

Wrap-Up & Takeaways

In This Notebook

In this notebook, you deployed an open-source Hugging Face model to Amazon SageMaker for real-time inference. You:

- Explored the Hugging Face model hub and selected a text classification model
- Used SageMaker's HuggingFaceModel class to deploy the model without Docker

- Sent test inputs using `boto3` directly
- Reviewed inference results and observed model behavior on varied input
- Cleaned up your endpoint and CloudFormation stack to avoid charges

---

This Workflow Reflects What Real ML Teams Do in Production

- Deploy pre-trained models to scalable inference endpoints
- Send payloads via APIs and automate feedback pipelines
- Monitor results and validate predictions against real user input
- Perform cost-aware infrastructure teardown when workloads are complete
- Use model hubs like Hugging Face to accelerate experimentation and delivery

What This Looks Like in Industry: MLOps + DevOps

While this notebook walks through deploying a single Hugging Face model manually, production-grade systems automate and extend this process through **MLOps** and **DevOps** practices.

MLOps Practices

- **Model Versioning**:
  Every model (and dataset) is versioned and stored — often using S3, Git, or tools like MLflow or SageMaker Model Registry.

- **Continuous Training Pipelines**:
  Training is automated via pipelines (e.g., SageMaker Pipelines, Airflow, or Kubeflow) triggered by data drift, retraining schedules, or performance degradation.

- **Endpoint Monitoring**:
  Production endpoints are monitored using tools like CloudWatch, Datadog, or Prometheus to detect latency spikes, input anomalies, or accuracy drops.

- **Shadow Testing + A/B Testing**:
  New models are deployed alongside existing ones to evaluate performance in live environments before being fully promoted.

DevOps Practices

- **Infrastructure as Code (IaC)**:
  Everything — IAM roles, S3 buckets, models, endpoints — is created via templates (e.g., CloudFormation, Terraform, or CDK).

- **CI/CD for Models**:
  GitHub Actions, CodePipeline, or Jenkins are used to deploy models automatically when code or config changes are pushed.

- **Environment Management**:
  Teams manage dev, staging, and prod environments separately using tags, branch-based workflows, or isolated accounts.

- **Cost and Resource Controls**:
  Autoscaling, scheduled shutdowns, and tagging help reduce cost and improve visibility across environments.

Final Thought

In this notebook, you walked through the **manual, educational path** of deploying and testing a Hugging Face model. In industry, these same steps are wrapped in automation, monitoring, and governance — enabling teams to scale GenAI reliably and securely.