# 50.055 Machine Learning Operations Hands On Session Notes

Authored by: Daniel Dahlmeier, January 2023.

## Introduction

Welcome to the hands-on sessions for the course 50.055 Machine Learning Operations. In these labs, you'll have the opportunity to apply your knowledge, explore concepts, and build practical skills. These hands-on sessions are designed for you to actively engage, try out various tasks, and gain a deeper understanding of the ML lifecycle. For each session, there are code examples provided but you are not limited to following these examples step by step. Instead, the code examples should be a starting point to further build and explore ML. We will be covering both what is now called “traditional” machine learning and “generative AI”.

You are allowed and encouraged to use AI tools, like ChatGPT, to generate code and solve technical problems but you have to understand the final results.

## Session 1: Train a linear model for sentiment classification

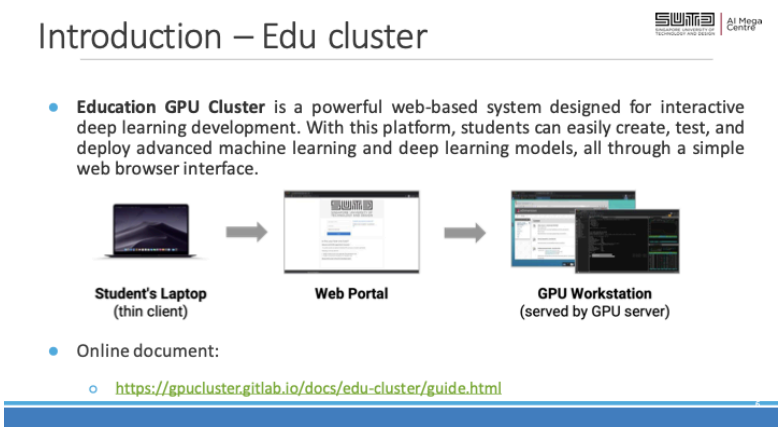
### Objective

In this session, you will train a linear classifier, a fundamental but powerful model in machine learning. We will apply it to sentiment classification. We will conduct the session on the SUTD education cluster.

### 1. Setup SUTD Education Cluster

We will use the SUTD Education Cluster for this course, in addition to AWS. Unlike AWS, there are no usage-based costs for using the SUTD cluster. Thus, the SUTD cluster can be an

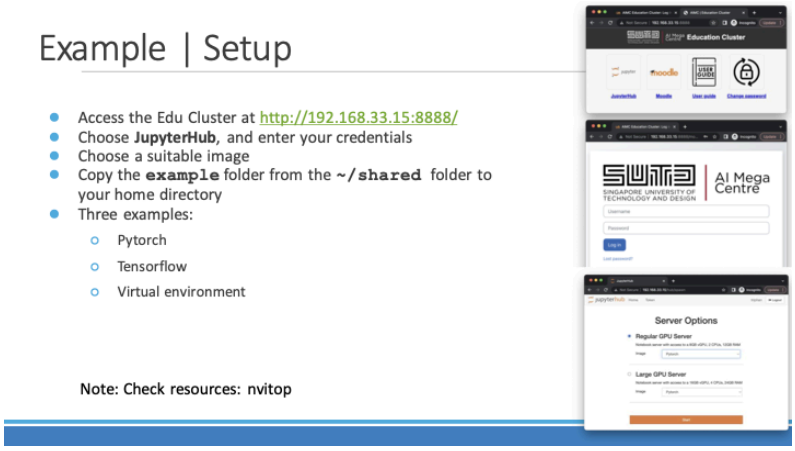
alternative to reduce costs. However, the SUTD cluster does not provide the large GPUs which are required for bigger LLMs and does not allow you to experiment with other AWS services like deployment endpoints.




Follow the instructions below to access the SUTD Education Cluster.

1. Your accounts on the cluster have already been created. Please login using the URL and username/password below. You need to be in the SUTD network to access the cluster (on campus or VPN).

- **Access URL**: http://192.168.33.15:8888/

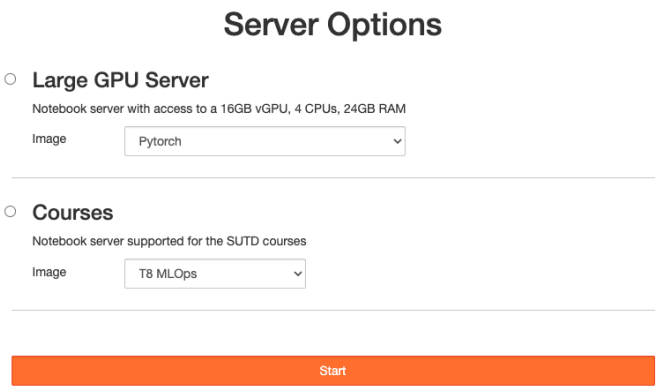



2. Choose JupyterHub, and enter your credentials

   **User ID**: your student ID (100XXXX)

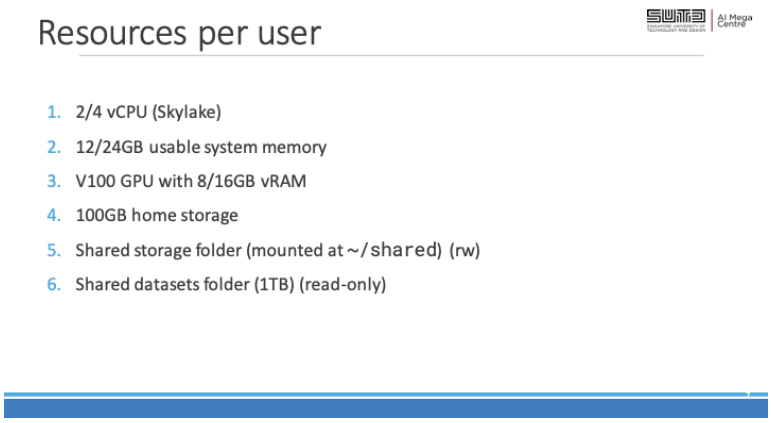
**Default password**: `AIcluster#2025` (needs to be changed after first login)

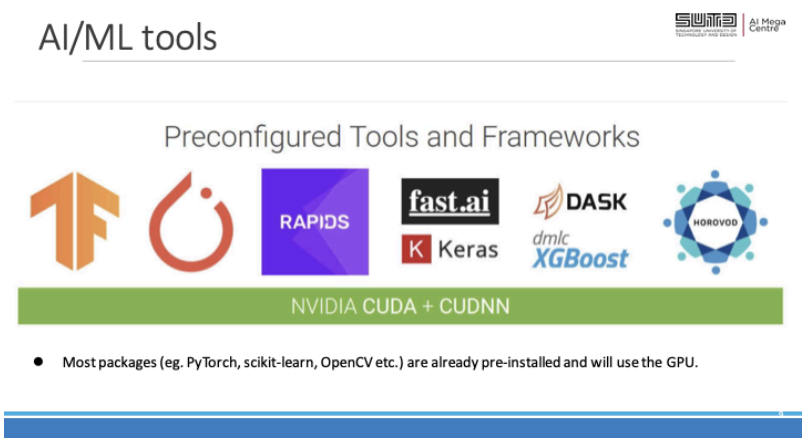
3. Choose the image **Course > T8 MLops.**




4. The technical specification and installed software stack of the cluster are as follows.

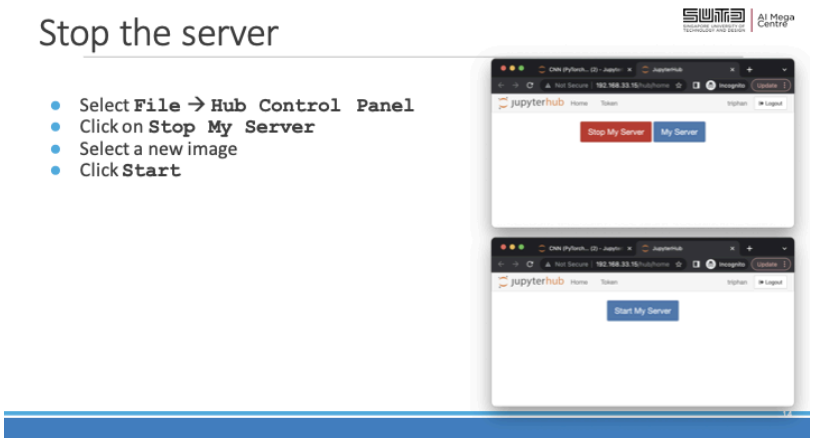
- 4 vCPU (Skylake)
- 24GB usable system RAM
- V100 GPU with 16GB vRAM






5. You are ready to start working on the exercise below now.

6. After finishing each hands-on session, please stop the notebook kernel and stop the server




For more details on the SUTD Education CLuster, see this user guide or the online documentation.

## 2. Refresh Github and Jupyter Notebooks Basics

If you are not yet familiar with Github and Jupyter Notebooks, or you have not used them in a long time, this is a good point to get to know these tools or to refresh your knowledge.

**Github Basics**
https://docs.github.com/en/get-started/quickstart/hello-world

**Notebook Basics**
https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/Notebook%20Basics.html

### 3. Start Jupyter notebook

Start a Jupyter notebook on the SUTD cluster.

### 4. Checkout Github repository

The code examples for this course are on Github. Check out this repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **01_train_linear model.ipynb** in the folder sutd-mlops-course-code.

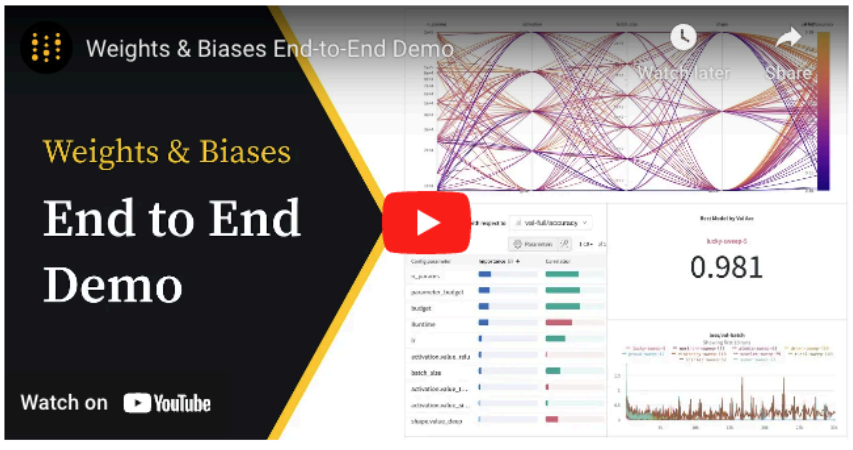
### 5. Create Weights & Biases account

Weights & Biases (wandb) is an experiment tracking tool for machine learning experiments. Create a free wandb account and go through the **Intro Notebook** and **Quickstart guide**
https://docs.wandb.ai/guides

Note
•
•

You

Are you a first-time user of W&B?




Start exploring W&B with these resources:

## 6. Create Huggingface account

Huggingface is a popular platform for foundation models and datasets
**Create a free  Huggingface account.**
https://huggingface.co/join

**Here is a quick tour of Huggingface dataset.**
https://huggingface.co/docs/datasets/index

## 7. Development environment

Run the pip install and import cells to make sure all dependencies are resolved.

Set up wandb for experimental tracking:

1. Login the wandb
2. Initiate the wandb with learning rate = 0.01

```
k = 1
wandb.init(
      # Set the project where this run will be logged
      project="sutd-mlops-project",
      # We pass a run name (otherwise it'll be randomly assigned, like sunshine-lollypop-10)
      name=f"experiment_session2_run_{k}",
      # Track hyperparameters and run metadata
      config={
      "learning_rate": 0.01,
      "architecture": "SGDClassifier",
      "dataset": "rotten_tomatoes",
      })
config = wandb.config
```

**Note**: if there is an error in wandb, try upgrading your wandb to version 0.16.2, and restart the kernel.

## 8. Download Rotten Tomatoes Movie Review Dataset

Download the Rotten Tomatoes movie review sentiment dataset.

```
# Download the Rotten Tomatoes dataset
dataset = load_dataset(config.dataset_name)

# print the first movie review and label
print(dataset["train"][0])
```

## Understanding Sentiment Classification

Sentiment classification involves assigning sentiment labels (e.g., positive, negative, neutral) to pieces of text, such as movie reviews. The linear classifier, a simple yet effective model, serves as an excellent starting point for such tasks. For a refresher on sentiment analysis you can check out the slides from Dan Jurafsky's course at Stanford:
https://web.stanford.edu/~jurafsky/slp3/slides/7_Sent.pdf

### 9. Data Exploration

Plot the distribution of positive and negative labels.

**Hint**: use the countplot from seaborn library,
https://seaborn.pydata.org/generated/seaborn.countplot.html

Question:

1. Is the dataset a balanced or unbalanced dataset?
2. How many classes in this dataset? (e.g., neutral, positive, negative)

**Questions**:

How many samples are there in the train, validation, and test set?

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 8530
    })
    validation: Dataset({
        features: ['text', 'label'],
        num_rows: 1066
    })
    test: Dataset({
```

```
    features: ['text', 'label'],
    num_rows: 1066
  })
})
```

What are the labels 0 and 1 corresponding to?

```
'label': ClassLabel(names=['neg', 'pos']
```

## 10. Feature engineering

Convert the text data into numerical features suitable for the linear classifier. For a start, we will create simple unigram features using the scikit-learn library.

**Hint**: use the CountVectorizer from scikit-learn library, https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

If you don't know scikit yet, check it out here: https://scikit-learn.org/stable/index.html, it is super helpful.


## 11. Train model

Instantiate a model. We are using a simple stochastic gradient classifier with logistic regression and fixed learning rate. Then train the model by calling model.fit()

**Hint**: use SGDClassifier from scikit-learn and set the loss to “log_loss”, and use the learning rate defined in wandb earlier.
https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

Check out the documentation carefully, take note of the differences between the parameter “learning_rate” and “eta”

**learning_rate** : ***str, default='optimal'***

The learning rate schedule:

- ‘constant’: `eta = eta0`
- ‘optimal’: `eta = 1.0 / (alpha * (t + t0))` where `t0` is chosen by a heuristic proposed by Leon Bottou.
- ‘invscaling’: `eta = eta0 / pow(t, power_t)`
- ‘adaptive’: `eta = eta0`, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss by tol or fail to increase validation score by tol if `early_stopping` is `True`, the current learning rate is divided by 5.

  *New in version 0.20:* Added ‘adaptive’ option

**eta0** : ***float, default=0.0***

The initial learning rate for the ‘constant’, ‘invscaling’ or ‘adaptive’ schedules. The default value is 0.0 as eta0 is not used by the default schedule ‘optimal’. Values must be in the range `(0.0, inf)`.

## 12. Test model

Make predictions on the test set and compute the accuracy. Log the accuracy to wandb together with a precision-recall curve.

Open wandb and inspect the logged metrics and plots.

Note:

- Use the plot_precision_recall from scikit-learn to plot the precision-recall curve
- Plot the curve in wandb
- call wandb.finish() after you finish your experiment, then a link will be provided to let you inspect your experiment.

## 13. Experimenting with hyperparameters

Experimenting with hyperparameters is an important aspect of model development. In the context of a linear classifier, hyperparameters might include learning rate, regularization strength, or the choice of optimization algorithm. Use wandb to track and compare the performance of different configurations.

Please repeat the run with k = 2, 3, 4, and other hyperparameters configurations (e.g., learning rate, loss function, penalty:

```
k = 1
wandb.init(
      # Set the project where this run will be logged
      project="sutd-mlops-project",
      # We pass a run name (otherwise it'll be randomly assigned, like sunshine-lollypop-10)
      name=f"experiment_session2_run_{k}",
      # Track hyperparameters and run metadata
      config={
      "learning_rate": 0.01,
      "loss": "log_loss",
      "penalty": "l2",
      "architecture": "SGDClassifier",
      "dataset": "rotten_tomatoes",
      })
config = wandb.config
```

Make sure that you define your model following the same configurations you set with wandb

```
model = SGDClassifier(
    loss = config.loss,
    penalty = config.penalty,
    learning_rate = 'constant',
    eta0 = config.learning_rate).fit(train_features, train_labels)
```

## 14. Cleanup

**Please clean up all resources.**

- Shut down the kernel and logout of the SUTD Education Cluster

## Conclusion

You are all set up to work with  the SUTD cluster. You have downloaded the first data set and trained a linear classifier for sentiment classification. Through experimentation and wandb logging, you gained experience with experiment tracking and hyperparameter tuning.

## Additional Resources

- Wandb Scikit Tutorial
- Scikit-learn

# Session 2: Train BERT classifier for sentiment classification in Jupyter Notebook

**Objective**

In this session, you will delve into fine-tuning a BERT model for classification. BERT is one of the most influential models of modern NLP. We will apply it to the sentiment classification problem from the last session. This will give us a direct comparison of "traditional" NLP and "modern NLP with foundation models. We will do the fine-tuning in an interactive Jupyter notebook environment.

## Understanding BERT

BERT is a pre-trained transformer encoder model which can be finetuned for downstream tasks, For an introduction to BERT, check out this Huggingfance blog:
https://huggingface.co/blog/bert-101

## 1. Start Jupyter notebook

Start a Jupyter notebook on the SUTD cluster.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **02_finetune_bert_notebook.ipynb** in the folder sutd-mlops-course-code.

### 3. Development environment, login to Weights & Biases

Run the first cells to install all required dependencies, import the necessary packages and log in to your wandb account. All this should feel familiar from the last hands-on session.

### 4. Data Preparation

Load the Rotten Tomatoes dataset again.

### ~~5. Feature engineering~~

Unlike the linear model, the BERT deep learning model learns the feature representations itself. Thus, we have no feature engineering step.

### 6. Tokenize data

A tokenizer separates text into individual tokens, usually a word or subset of a word. It is important to use the correct tokenizer. Luckily Huggingface makes it easy to load the correct tokenizer for a model with the AutoTokenizer.from_pretrained() method.
We will be using the DistilBERT model and therefore need to load its associated tokenizer.

```
model_name = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

## 7. Subsample data

Training deep learning models is a lot slower than training linear models. To reduce the time for this hands-on exercise, we select a random subsample of the training, evaluation, and test data.

## 8. Load model

Load the DistilBERT model from Huggingface model hub. DistilBERT is a smaller version of BERT which requires less memory and is faster to train.

You can learn more about DistilBERT here: https://huggingface.co/papers/1910.01108

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)
```

## 9. Train model

We use the Hugginface Trainer class to start the model training with the required training arguments and evaluate . We log training metrics to wandb.

## 10. Test model

Make predictions on the training, validation and test set and compute the accuracy.

Open wandb and inspect the logged metrics and plots of the experiment.

## 11. Experimenting with hyperparameters

The Huggingface Trainer class has a lot of hyperparameters. If you have time, explore different parameters and their effect, for example number of epochs, batch size, or learning rate

If you have time, re-run the experiment with the full data set and with the original BERT model instead of DistillBERT.

### 12. Cleanup

**Please clean up all resources.**

- Shut down the kernel and logout of the SUTD Education Cluster

## Conclusion

This session provided hands-on experience with finetuning a BERT classifier for sentiment classification. Now you can finetune state-of-the-art NLP models.

## Additional Resources

- BERT research paper
- DistilBERT research paper

# Session 3: Accelerated Inference with Optimum and Transformers Pipelines

## Objective

In this session, you will optimize a trained Huggingface transformer model to make it ready for deployment. We will use the Optimum library and ONNX to reduce model size and improve latency of a pre-trained Question Answering model. The session is based on the excellent blog by Philipp Schmid from Huggingface.


## 1. Start Jupyter notebook

Start a Jupyter notebook on the SUTD cluster.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **03_optimize_onnx.ipynb** in the folder sutd-mlops-course-code.

## 3. Development environment

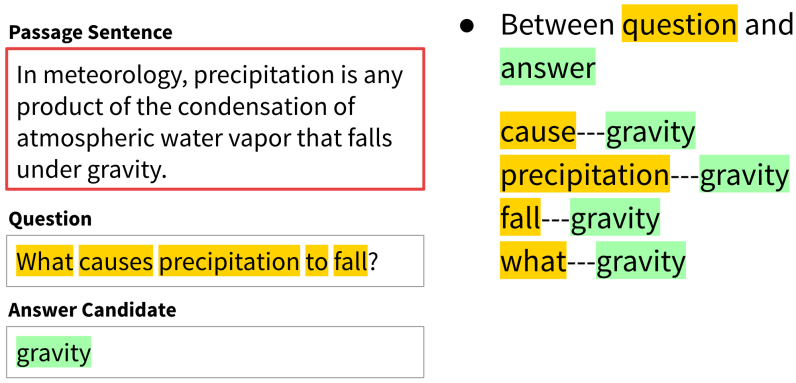
Run the first notebook cell to install the optimum package.

## 4. Load RoBERTa SQUAD model

In this session, we are not training our own model. Instead we will be using a pre-trained RoBERTa mode for question answering from Huggingface model hub.

RoBERTa is based on BERT but introduces a couple of improvements in the way it was trained. Read theis blog about RoBERTa: https://huggingface.co/docs/transformers/model_doc/roberta

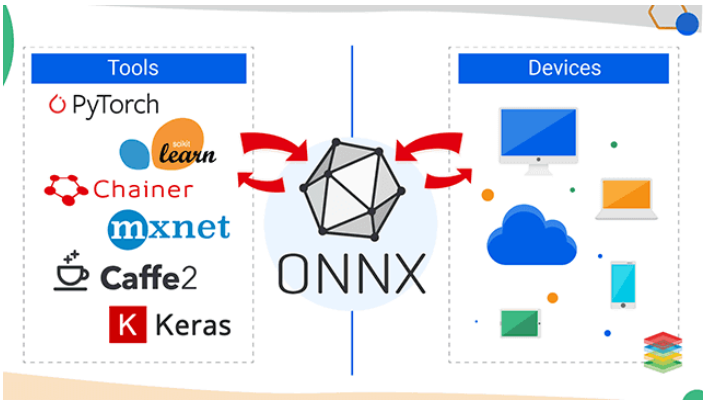
The model has been trained to perform question answering on SQuAD, the Stanford Question Answering Dataset. SQuAD has been a very popular NLP benchmark. Check out the SQuAD website and browse a few examples: https://rajpurkar.github.io/SQuAD-explorer/




## 5. Convert model to ONNX

We load the vanilla model and convert it to the ONNX format. ONNX stands for Open Neural Network Exchange. It provides an open interchange data format for neural networks. This allows developers to convert their models from PyTorch, TF or other frameworks into a common format. This is typically done to make it easier to deploy models. Check out the ONNX project here: https://onnx.ai/get-started.html

```
from optimum.onnxruntime import ORTModelForQuestionAnswering

model = ORTModelForQuestionAnswering.from_pretrained(model_id, from_transformers=True)
```




## 6. Run question answering example

We create a pipeline and run a simple question answering example, just to confirm that it works.

```
optimum_qa = pipeline(task, model=model, tokenizer=tokenizer, handle_impossible_answer=True)
prediction = optimum_qa(question="What's my name?", context="My name is Philipp and I live in Nuremberg.")

print(prediction)
```

## 7. Optimize model

Use the optimum library to apply graph optimization, such as operator fusion and constant folding to accelerate latency and inference. Then save the model to disk and run the simple question answering example again. It should run faster already.

```
# create ORTOptimizer and define optimization configuration
optimizer = ORTOptimizer.from_pretrained(onnx_path)

optimization_config = OptimizationConfig(optimization_level=99) # enable all optimizations

# apply the optimization configuration to the model
optimizer.optimize(save_dir=onnx_path, optimization_config=optimization_config)
```

## 8. Quantize the model

Next, we apply quantization to the model weights to reduce model size.

```
# create ORTQuantizer and define quantization configuration
quantizer = ORTQuantizer.from_pretrained(onnx_path, file_name="model_optimized.onnx")
qconfig = AutoQuantizationConfig.avx512_vnni(is_static=False, per_channel=True)

# apply the quantization configuration to the model
quantizer.quantize(save_dir=onnx_path, quantization_config=qconfig)
```

Then we compare the model size of the vanilla model and the quantized model. The quantized model is only about ¼ of the original size.

```
size = os.path.getsize(onnx_path / "model.onnx")/(1024*1024)
print(f"Vanilla Onnx Model file size: {size:.2f} MB")
size = os.path.getsize(onnx_path / "model_optimized_quantized.onnx")/(1024*1024)
print(f"Quantized Onnx Model file size: {size:.2f} MB")

# Vanilla Onnx Model file size: 473.51 MB
```

```
# Quantized Onnx Model file size: 119.15 MB
```

Again, create a pipeline and run the question answering example. It should load even faster.

## 9. Evaluate performance

How good is the optimized and quantized model compared to the original? Load the SQuAD2 dataset to compare the performance of the orignal and optimized-quantized model. Because the dataset is large, we just load 10% of the validation dataset. Feel free to play with larger portions of the dataset or the full dataset later.

The result shows that the drop in performance is very small. We get more than 98% of the original performance with a model that is a quarter the size.

*# vanilla model: exact=81.12889637742207% f1=83.27089343306695%*
*# optimized model: exact=81.12889637742207% f1=83.27089343306695%*
*# quantized model: exact=79.78096040438079% f1=82.22196132052908%*

## 10. Evaluate speed

How fast is the optimized and quantized model compared to the original? We run a simple latency test with a slightly longer version of the original sample question. The code runs a few warmup loops to take care of effects like model loading, then executes the QA example 100 times while measuring the latency.

The result shows that optimized and quantized model is about 30% faster than the original. Note that the exact numbers depend on many factors, such the exact hardware, and therefore your numbers can look different. You should, however, see some speedup.

*# Vanilla model Average latency (ms) 126*
*# Optimized & Quantized model Average latency (ms) 89*

## 11. Cleanup

**Please clean up all resources.**

- Shut down the kernel and logout of the SUTD Education Cluster

# Conclusion

Optimizing a model before deployment can be important to achieve low latency and cost-efficient deployment. This session provided the first experience with how to optimize models with Optimum and ONNX.

## Additional Resources

- Huggingface ONNX tutorial: https://huggingface.co/blog/optimum-inference
- ONNX tutorials: https://github.com/onnx/tutorials
- RoBERTa paper: https://arxiv.org/abs/1907.11692
- SQuAD paper: https://arxiv.org/pdf/1606.05250v3.pdf

## Session 4: Setup AWS Environment and label data

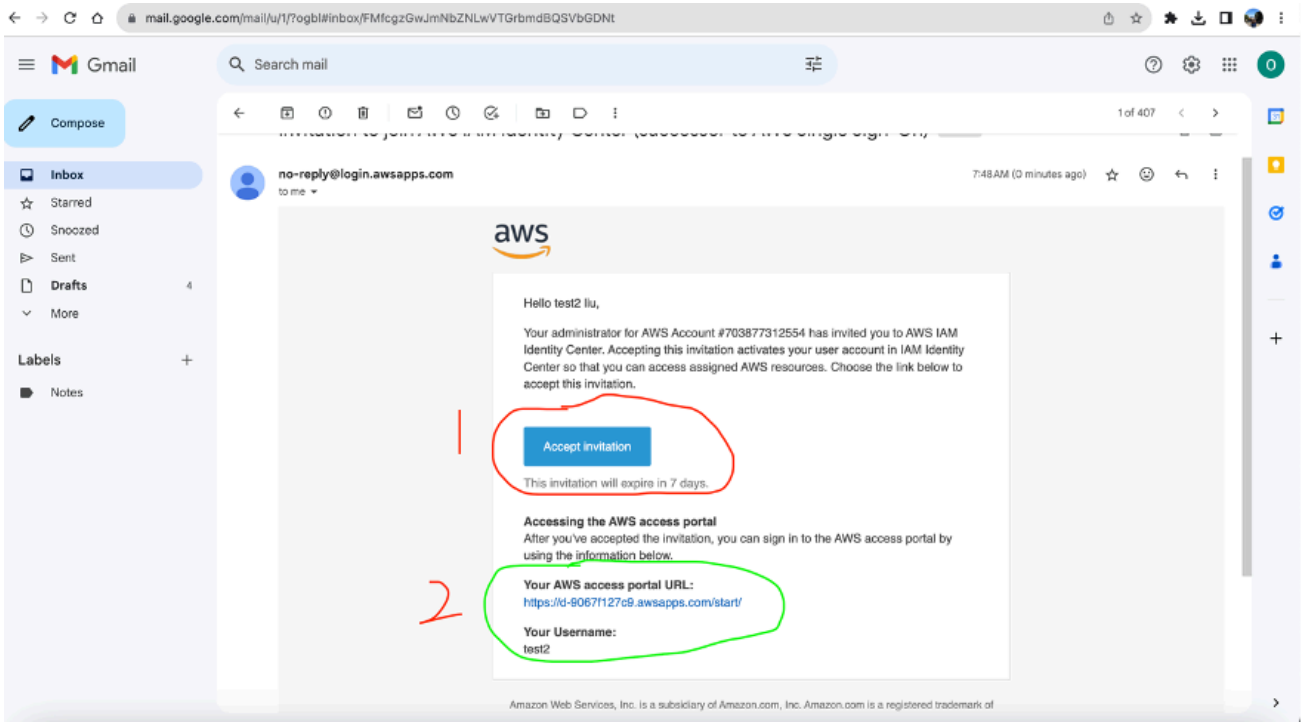
## Objective

In this session, you will set up an AWS environment and complete a data labeling job using Sagemaker Groundtruth. Creating high-quality data assets with good labels is essential to the success of ML projects.

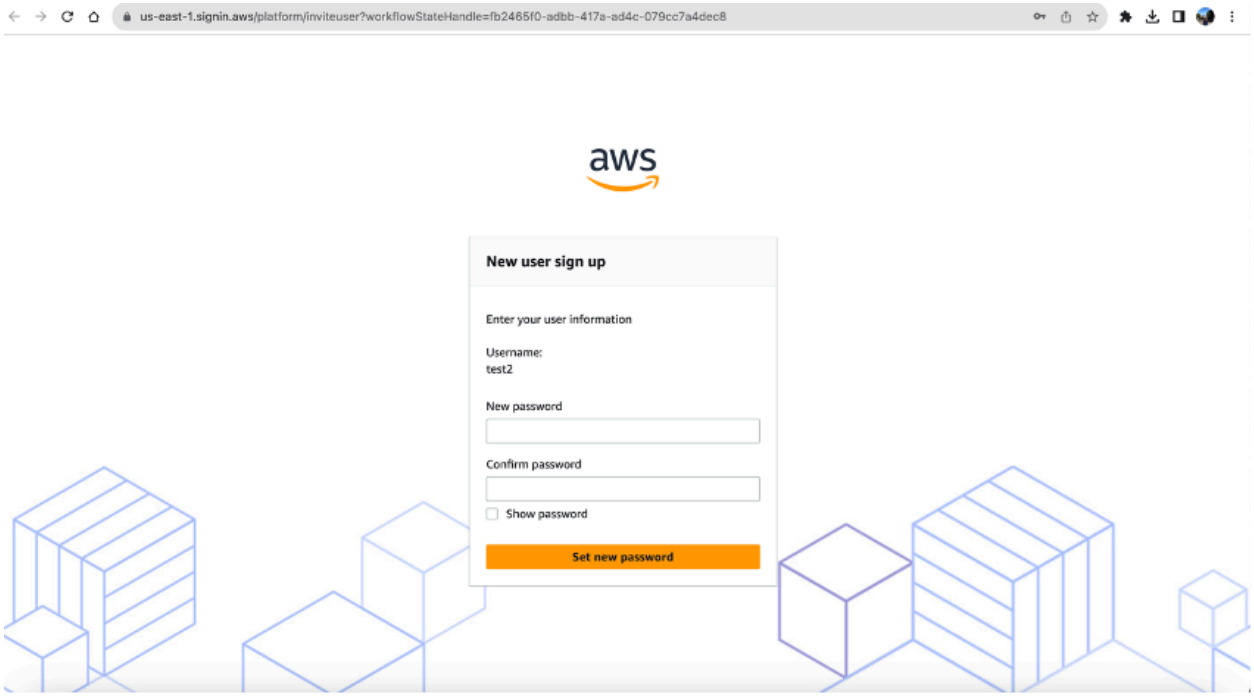
### 1. Login to AWS

Please kindly follow below steps to have your account set up for using AWS Sagemaker Studio.

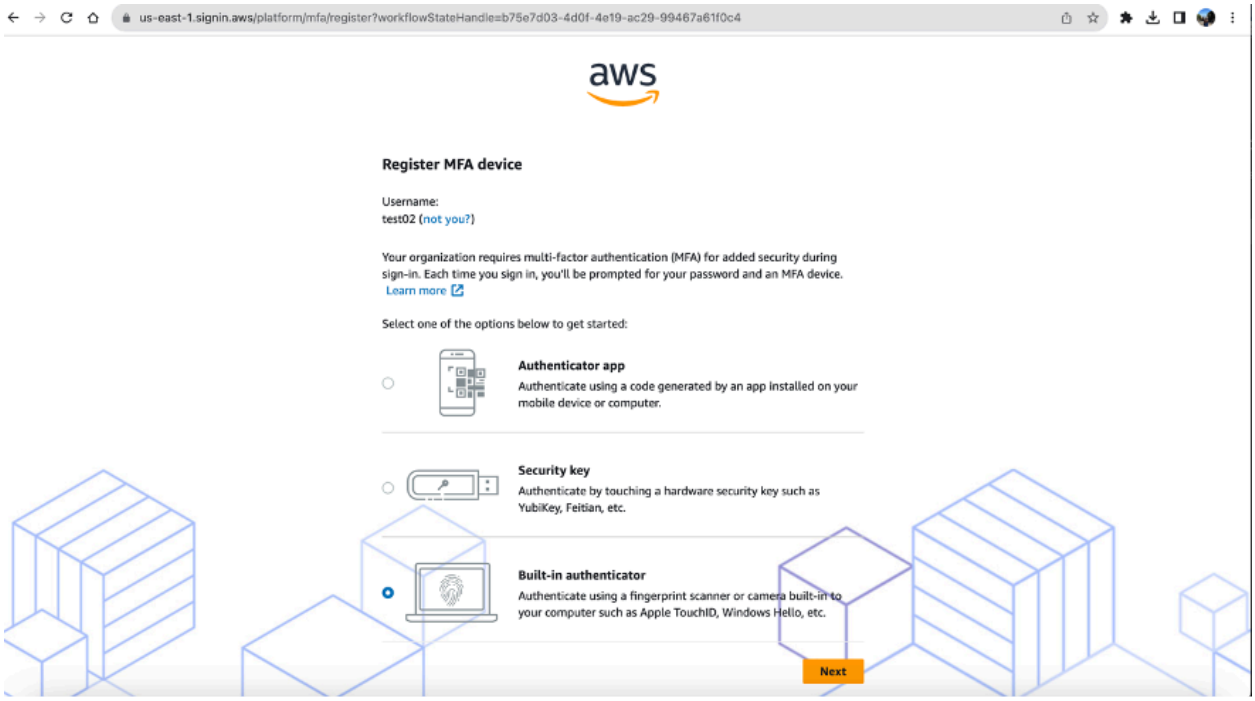
1. You will receive an invitation email as shown below in the picture in your school email account. (Please check your email and let us know if you haven't received)

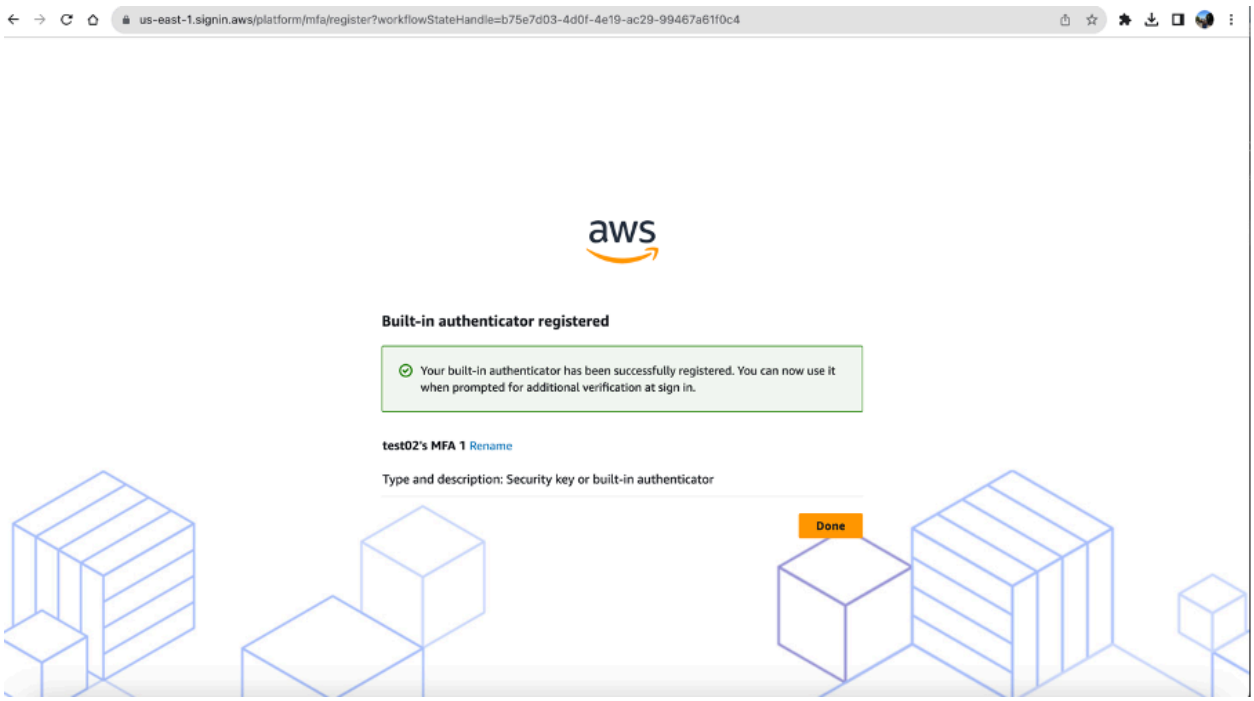


2. By clicking on Accept Invitation, you will be asked to set a new password.

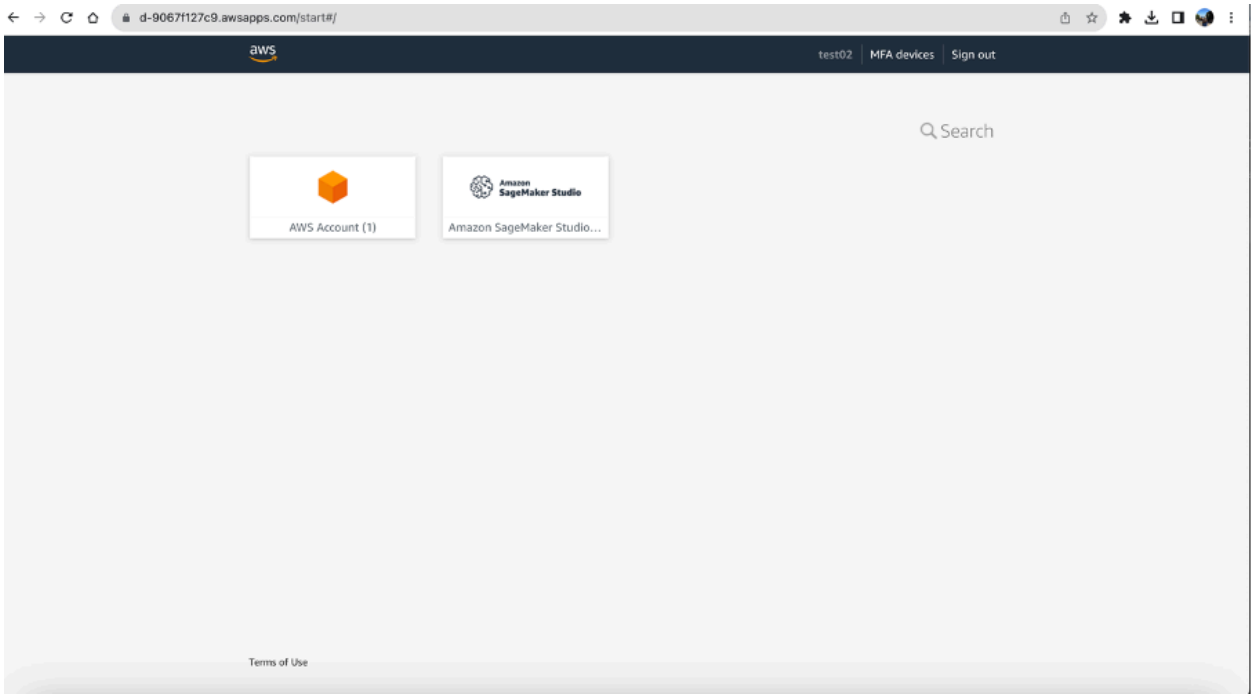



3. Then you will need to set up the MFA (you can choose whichever one works better for you).



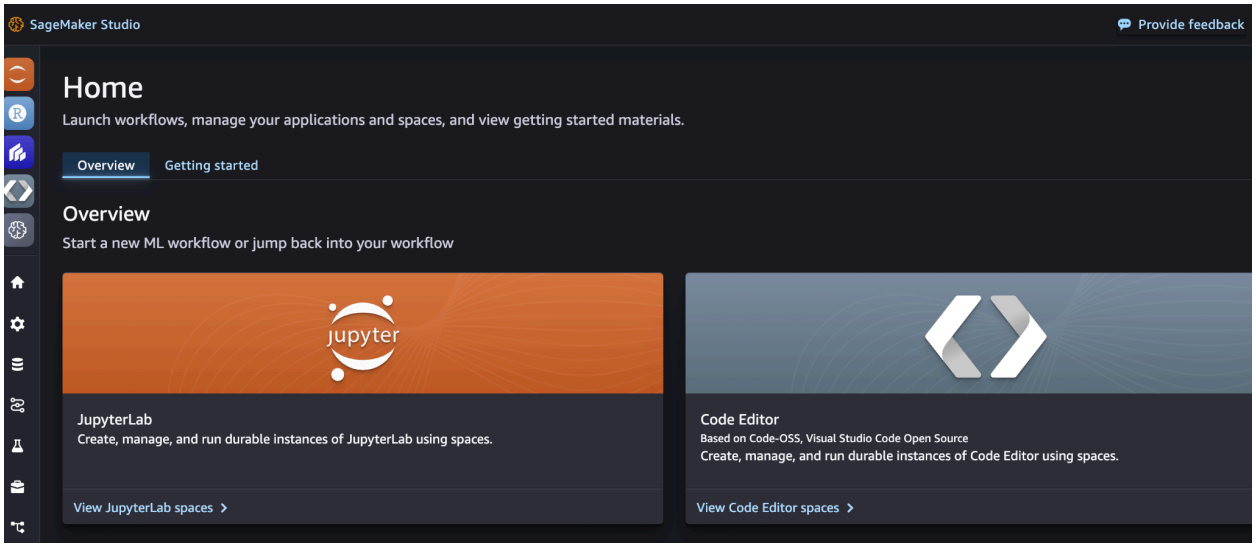

4. You should be directed to the page with only Amazon Account and Sagemaker Studio application as shown below. You can use your username and password to log in later via the portal link https://d-9067f127c9.awsapps.com/start/ which can also be found in your invitation email (as highlighted in step 1 picture 1 above).



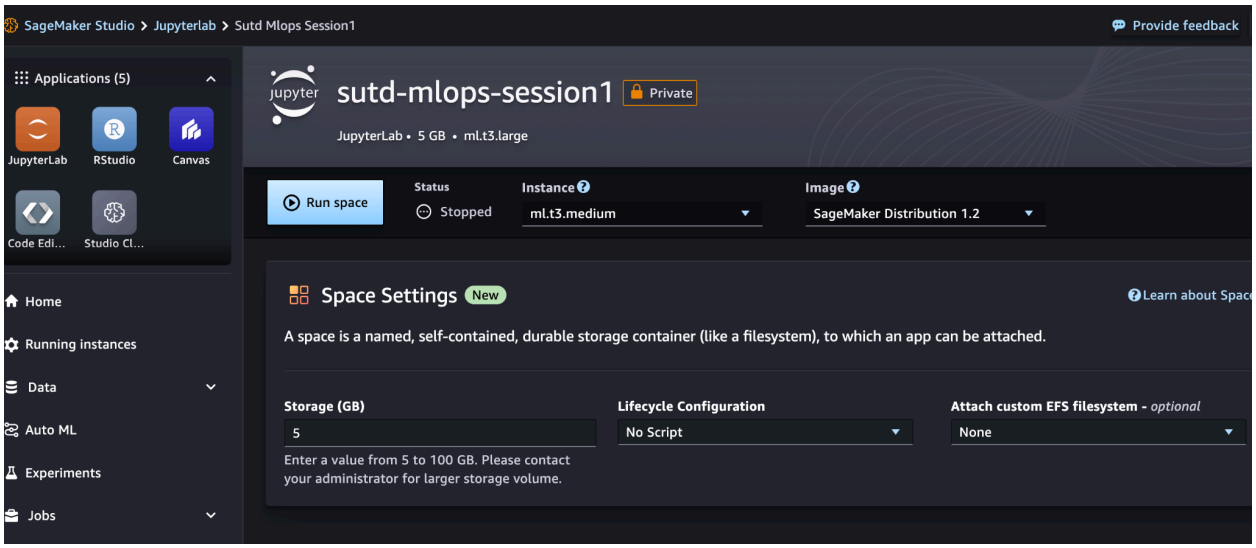

## 2. AWS Sagemaker Studio

Now that you have your account set up, you can start using Sagemaker Studio.

1. Open up Sagemaker Studio.



2. Create a JupyterLab space with name **sutd-mlops-session-<your student-id>** and default configuration: **ml.t3.medium** compute instance and **5GB** storage. Open the JupyterLab notebook.




## 3. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **04_sagemaker.ipynb** in the folder sutd-mlops-course-code.

## 4. Download Rotten Tomatoes Movie Review Dataset

Download the Rotten Tomatoes movie review sentiment dataset.

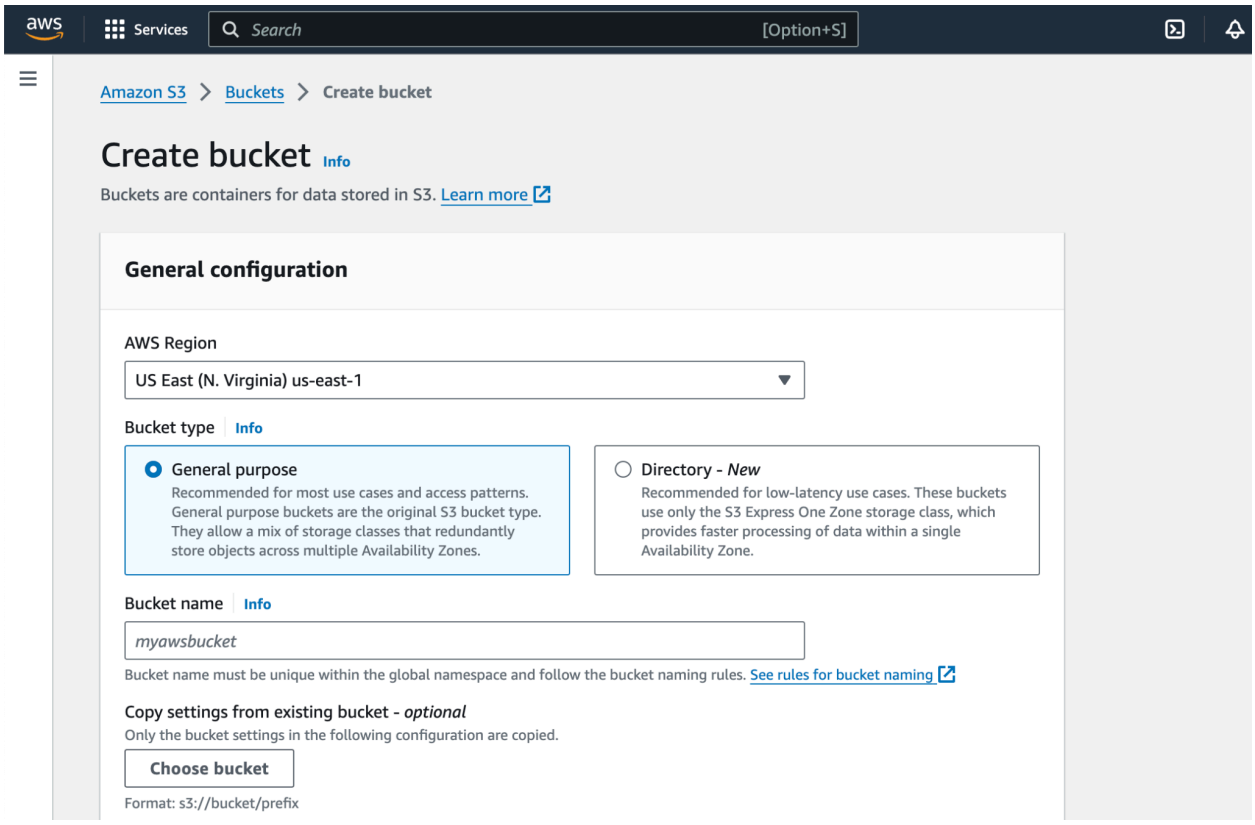
```
# install huggingface datasets
! pip install datasets

# load dependencies
from datasets import load_dataset
```

```
# Download the Rotten Tomatoes dataset
rotten_tomatoes_dataset = load_dataset(“rotten_tomatoes”)

# print the first movie review and label
print(rotten_tomatoes_dataset["train"][0])
```

## 5. Create an S3 bucket and upload data

We will use  Sagemaker Ground Truth to label the dataset. We need to upload the data to S3 to make it accessible to Ground Truth.

Navigate to the AWS console and open the S3 service.  Create a new bucket with the name **<your_user_id>-sagemaker-groundtruth.**



Now go back to the JupyterLab. Shuffle the dataset with the fixed seed 42 and then select the first 20 reviews.
Upload these 20 reviews without the labels (i.e just the review text) to the S3 bucket you just created, each as a separate text file.

**Hints**:

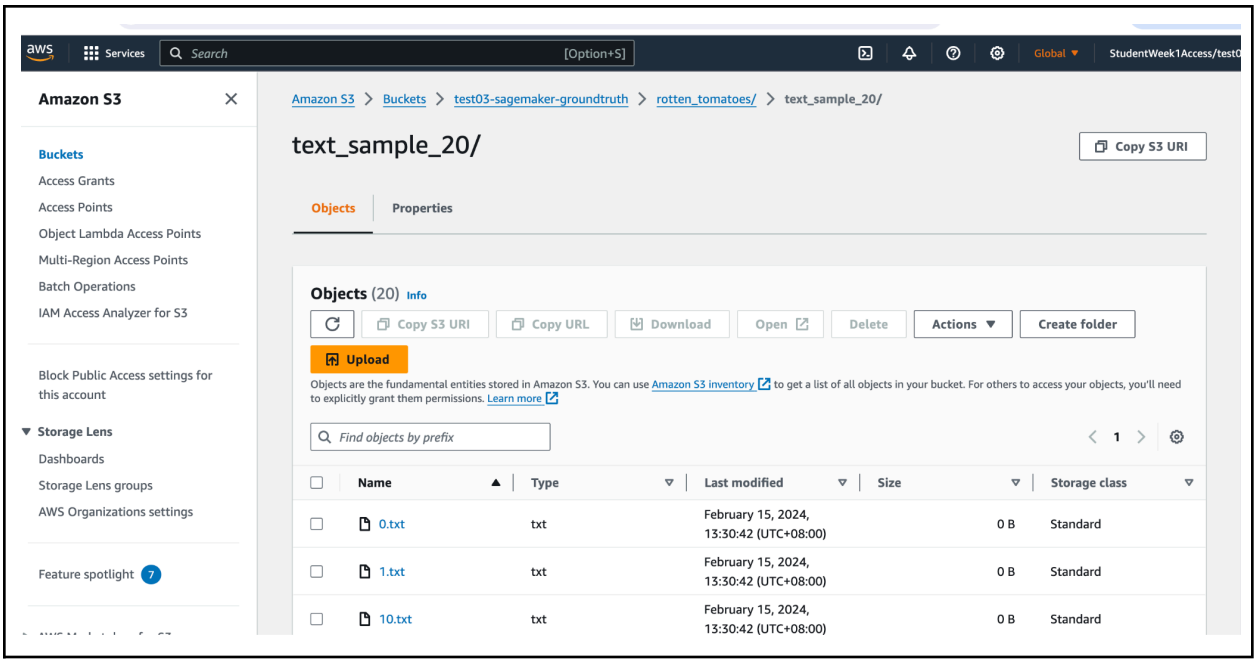
- Extract 20 sample texts from training set, write the data to .txt file
- Identify the bucket (make sure the bucket is located in the same region as the SageMaker
- Use boto3 to upload the data to the s3 bucket.

Reference:
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3/client/upload_file.html#

**Example**:

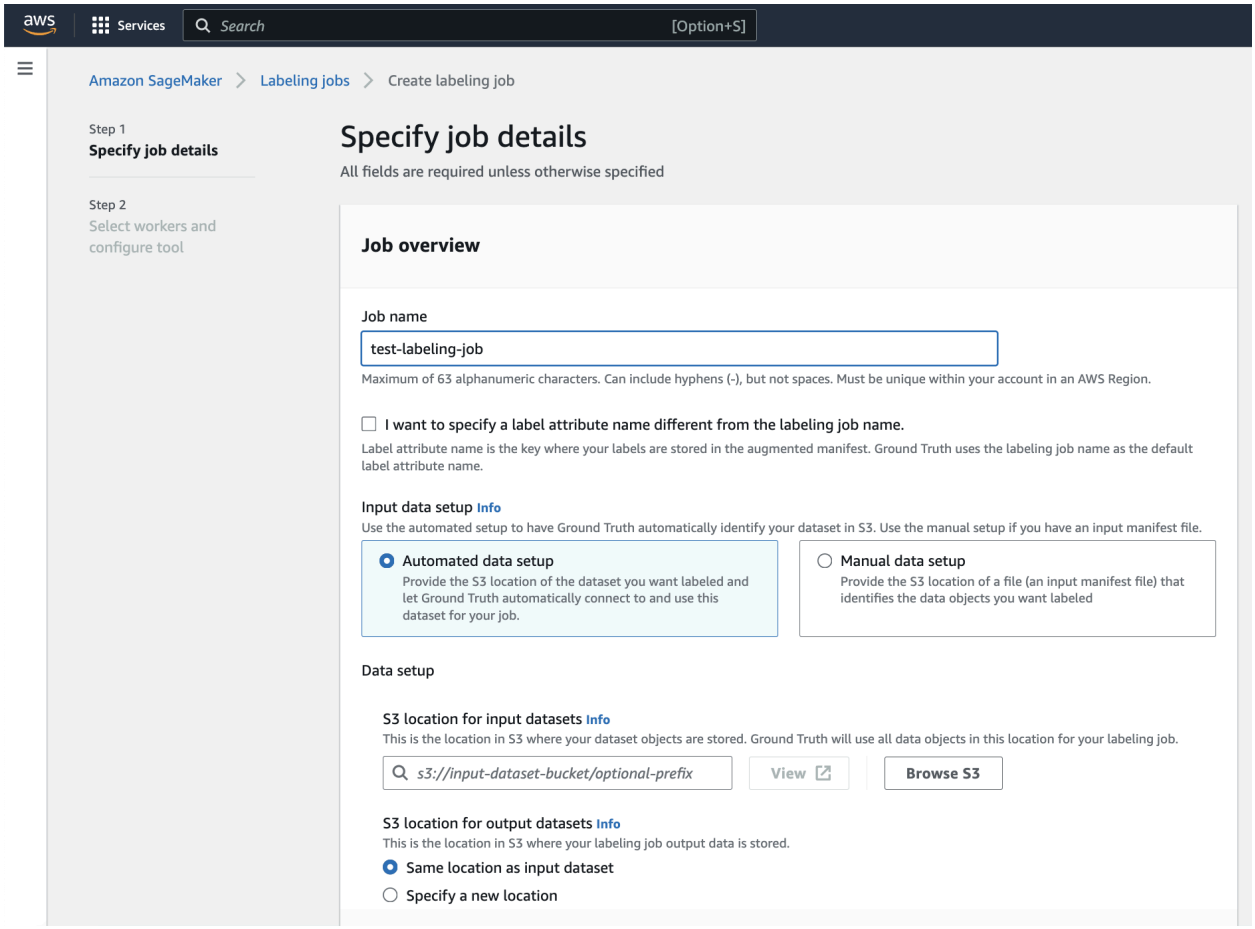
```
# select random 20 samples of movie review data
text_list = rotten_tomatoes_dataset['train'].shuffle(seed=42)[0:20]

# create directory
! mkdir -p text_sample_20


import sagemaker
import boto3

# create bucket
s3 = boto3.resource('s3')
bucket = s3.Bucket('<your_user_id>-sagemaker-groundtruth')  # you need to change the bucket name

# save into txt file and upload to S3
for k, sample_textin enumerate(text_list['text']):
    with open(f'text_sample_20/{k}.txt', 'w') as f:
        f.write(sample_text)
    bucket.upload_file(f'text_sample_20/{k}.txt', f'rotten_tomatoes/text_sample_20/{k}.txt')
```

If you did the above correctly, you should be able to see your 20 files in the AWS S3 bucket, as follows:




## 6. Label data using AWS Sagemaker Ground Truth

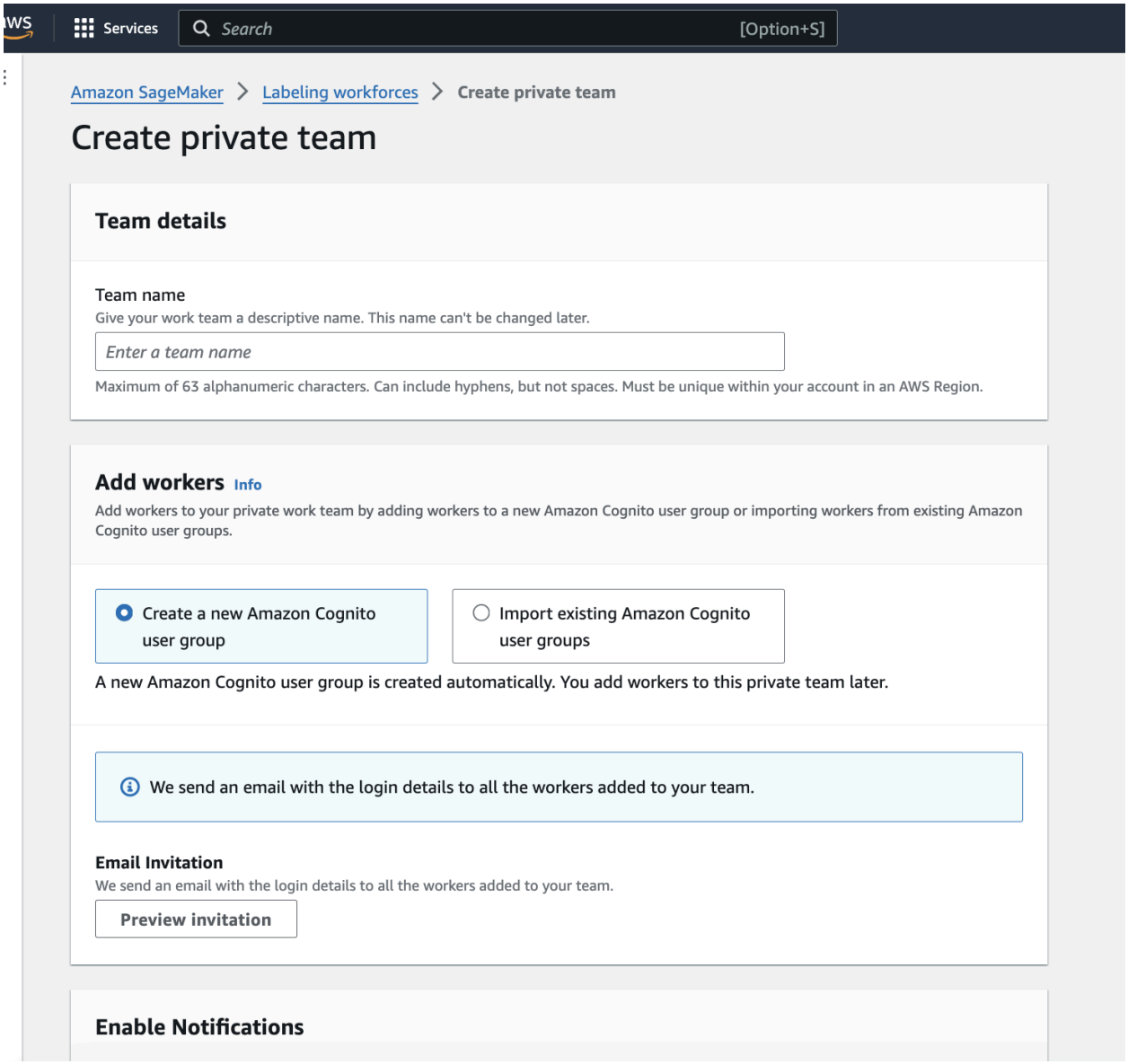
Go to the AWS console and open Sagemaker. Navigate to Ground Truth. Next, create a labeling job in Sagemmaker Ground Truth




You can learn more about Ground Truth here.
https://aws.amazon.com/tutorials/machine-learning-tutorial-label-training-data/
(**Note**: this tutorial uses images as the input data, but for our lab, we will use text from the rotten_tomatoes dataset as the input.)

You can skip step 1, 2 and 5 in this tutorial, and use the Jupyter studio you have set up previously to complete this tutorial. Adjust the tutorial where necessary to use the Rotten Tomatoes review texts, rather than the caltech-101 images)

**Assign yourself as the label workforce.**
Instead of using mechanical turk as a labeling workforce, you will be doing the labeling yourself. Create a new private team in Ground Truth and invite yourself as a worker using your email address.

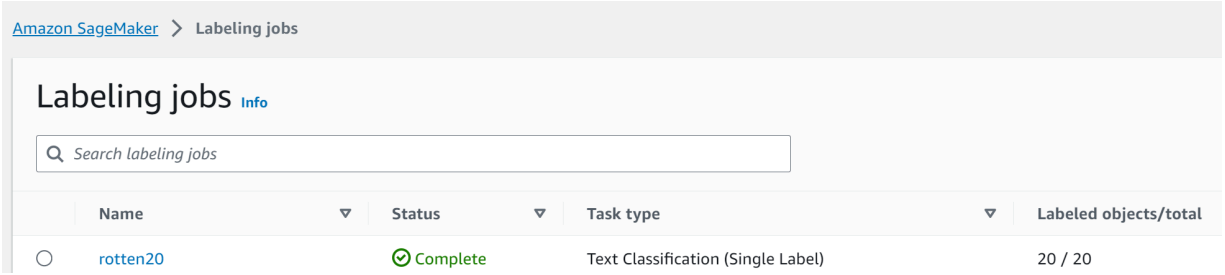



Then look out for the invite email to do the labeling job. Log in and complete the labeling work. Do NOT look at the original labels when performing the job!

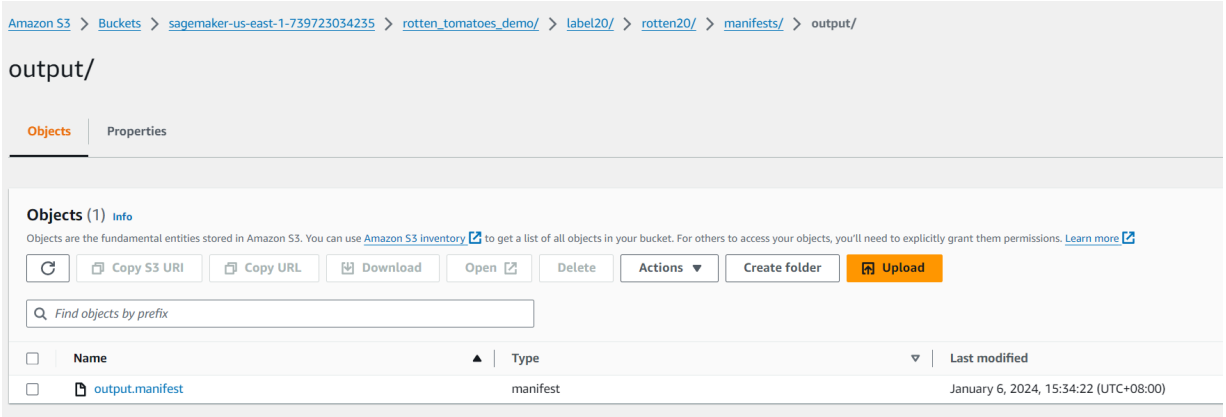
How difficult was the labeling job? Did you encounter examples that were ambiguous or unclear?

Afterwards go back to Sagemaker studio and download the results back to Sagemaker Studio.

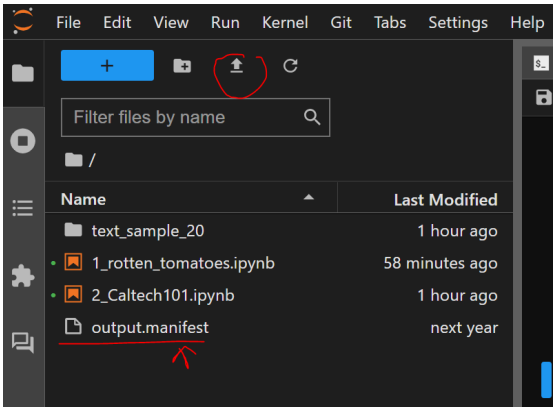
| You can check the status to see if the labeling job is completed: |
|---|




Locate your labeling output based on your initial labeling configuration (e.g., if you specify the output folder as “rotten_tomatoes_demo/label20”, you should be able to locate your output manifest file in “rotten_tomatoes_demo/label20/rotten20/manifests/output/output.manifest”




You can download the output.manifest and upload it to SageMaker




The manifest file consists of 20 json objects. Each json object defines the source (i.e., the text input), and the class-name (inside the metadata). See example below. Use the file operations and json functions to create a python function that organizes the data following the similar data format when you load the data from hugging face. **Hint**: use dict

```
import json

with open("output.manifest", 'r') as f:
    data20 = f.read()

temp = data20.split('\n')
json.loads(temp[0])
```

```
{'source': 'the rock is destined to be the 21st century\'s new " conan "
teven segal .',
 'rotten20': 0,
 'rotten20-metadata': {'class-name': 'positive',
  'job-name': 'labeling-job/rotten20',
  'confidence': 0,
  'type': 'groundtruth/text-classification',
  'human-annotated': 'yes',
  'creation-date': '2024-01-06T07:26:43.006570'}}
```

Example function:

```
def organized_data(data20):
    import json

    temp = data20.split('\n')
    self_annotated_data_20 = {'text': [], 'label': []}
    for tt in temp:
        if tt == '': continue
        jo = json.loads(tt)
        self_annotated_data_20['text'].append(jo['source'])
        self_annotated_data_20['label'].append(1 if jo['rotten20-metadata']['class-name'] == 'positive' else 0)
    return self_annotated_data_20
```

## 7. Compute annotator agreement

Load the first 20 examples of the movie reviews with labels plus your own  annotations
Compute the agreement between the original labels and your labels. Review the examples where you disagree. Who was correct, you or the original label?

Now you have your own labeled data, compare it with the original label from the rotten_tomatoes dataset.  **Note**: the first cell shows your own labeled data, and the second cell shows the original data

```
self_annotated_data_20 = organized_data(data20)
self_annotated_data_20
```

```
{'text': ['the rock is destined to be the 21st century\'s new " conan " and that he\'s going to make a splash even greater than arnold schwarzene
  'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge that a column of words cannot adequately describe co-wri
  'this is a film well worth seeing , talking and singing heads and all .',
  'what really surprises about wisegirls is its low-key quality and genuine tenderness .',
  '( wendigo is ) why we go to the cinema : to be fed through the eye , the heart , the mind .',
  'one of the greatest family-oriented , fantasy-adventure movies ever .',
  'ultimately , it ponders the reasons we need stories so much .',
  "an utterly compelling 'who wrote it' in which the reputation of the most famous author who ever lived comes into question .",
  'illuminating if overly talky documentary .',
  'a masterpiece four years in the making .',
  "the movie's ripe , enrapturing beauty will tempt those willing to probe its inscrutable mysteries .",
  'offers a breath of the fresh air of true sophistication .',
  'effective but too-tepid biopic',
  'if you sometimes like to go to the movies to have fun , wasabi is a good place to start .',
  "emerges as something rare , an issue movie that's so honest and keenly observed that it doesn't feel like one .",
  'the film provides some great insight into the neurotic mindset of all comics -- even those who have reached the absolute top of the game .',
  'offers that rare combination of entertainment and education .',
  'perhaps no picture ever made has more literally showed that the road to hell is paved with good intentions .',
  "steers turns in a snappy screenplay that curls at the edges ; it's so clever you want to hate it . but he somehow pulls it off .",
  'take care of my cat offers a refreshingly different slice of asian cinema .'],
 'label': [1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]}
```

```
original_data_20 = rotten_tomatoes_dataset['train'][0:20]
original_data_20
```

```
{'text': ['the rock is destined to be the 21st century\'s new " conan " and that he\'s going to make a splash even greater than arnold schwarzene
  'the gorgeously elaborate continuation of " the lord of the rings " trilogy is so huge that a column of words cannot adequately describe co-wri
  'effective but too-tepid biopic',
  'if you sometimes like to go to the movies to have fun , wasabi is a good place to start .',
  "emerges as something rare , an issue movie that's so honest and keenly observed that it doesn't feel like one .",
  'the film provides some great insight into the neurotic mindset of all comics -- even those who have reached the absolute top of the game .',
  'offers that rare combination of entertainment and education .',
  'perhaps no picture ever made has more literally showed that the road to hell is paved with good intentions .',
  "steers turns in a snappy screenplay that curls at the edges ; it's so clever you want to hate it . but he somehow pulls it off .",
  'take care of my cat offers a refreshingly different slice of asian cinema .',
  'this is a film well worth seeing , talking and singing heads and all .',
  'what really surprises about wisegirls is its low-key quality and genuine tenderness .',
  '( wendigo is ) why we go to the cinema : to be fed through the eye , the heart , the mind .',
  'one of the greatest family-oriented , fantasy-adventure movies ever .',
  'ultimately , it ponders the reasons we need stories so much .',
  "an utterly compelling 'who wrote it' in which the reputation of the most famous author who ever lived comes into question .",
  'illuminating if overly talky documentary .',
  'a masterpiece four years in the making .',
  "the movie's ripe , enrapturing beauty will tempt those willing to probe its inscrutable mysteries .",
  'offers a breath of the fresh air of true sophistication .'],
 'label': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

What is the accuracy of your own annotation?

## 8. Explore Sagemaker

Explore AWS Sagemaker, a fully managed service that covers the entire machine learning workflow. Set up your Sagemaker environment for future machine learning tasks.
We will be using Sagemaker as the main ML development environment for this course.

Watch this short introduction to AWS Sagemaker: **Introduction to Amazon SageMaker Studio**
https://www.youtube.com/watch?v=YcJAc-x8XLQ


Learn the basics of Amazon Sagemaker, including how to create and manage JupyterLab notebooks.

Here are some tutorials to get started on AWS Sagemaker. Go through the tutorial **Build an ML model locally.**

https://aws.amazon.com/sagemaker/getting-started/

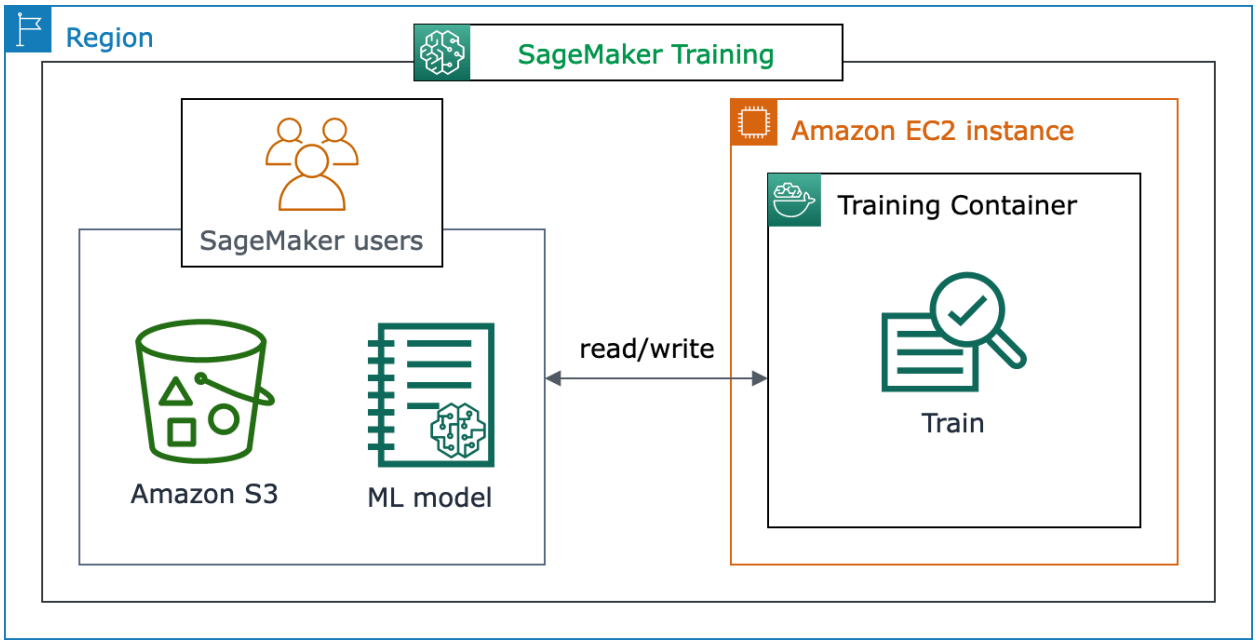
**IMPORTANT: avoid cost overruns**

- Using AWS resources creates costs
- Leaving instances running can create significant cost overruns
- **Switch off** all instances, training jobs and notebooks after each session

# Session 5: Train BERT classifier for sentiment classification in Amazon Sagemaker

## Objective

In this session, you will repeat fine-tuning a BERT model for classification but instead of working in an interactive Jupyter notebook environment, we will use Amazon Sagemaker.




## 1. Login to AWS and start Jupyter notebook in Sagemaker Studio

Start a JupyterLab space with a default **ml.t3.medium** instance on AWS Sagemaker Studio.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **05_finetune_bert_sagemaker_cluster.ipynb** in the folder sutd-mlops-course-code.

## 3. Development environment, Wandb, Data Preparation, tokenization, sampling

Run the first few notebook cells to do the by-now-familiar installation of dependencies, login to wandb.

Load the Rotten Tomatoes dataset and run the familiar steps of data tokenization and sampling.

Note that we are initializing an **S3 bucket**. S3 is a cloud storage service by AWS

**What is Amazon S3**

https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

## 4. Upload training and evaluation data to S3

To make the data to the AWS Sagemaker training job, we need to upload the training and validation data to our prepared S3 bucket

```
# save train_dataset to s3
training_input_path = f's3://{sess.default_bucket()}/{s3_prefix}/train'
train_dataset.save_to_disk(training_input_path)

# save validation to s3
```

```
test_input_path = f's3://{sess.default_bucket()}/{s3_prefix}/validation'
test_dataset.save_to_disk(test_input_path)
```

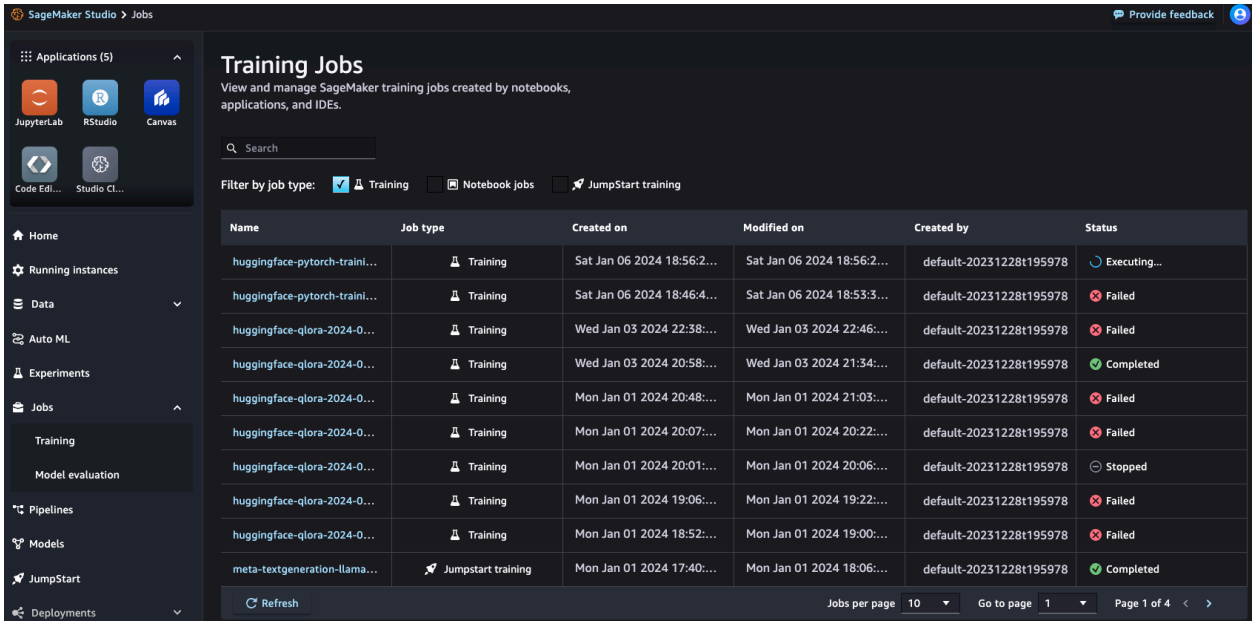
## 5. Train the model on AWS

Unlike when we work in an interactive notebook, the training will happen in the AWS cloud. Therefore we need to provide a script with the training code. For this example, the training code is already prepared in **train_sagemaker.py.** Have a look at the code.

Next, we create an estimator class which configures the training job. Sagemaker provides a special Huggingface class for training Huggingface models. We set the training hyperparameters, which includes the name of the base model,  and submit the training job. This will create a training job in AWS Sagemaker, spin up the required instance, load a Docker container with the training code, load the data and execute the job.

```
from sagemaker.huggingface import HuggingFace

# hyperparameters, which are passed into the training job
hyperparameters={'epochs': 1,
            'train_batch_size': 32,
            'learning_rate': 2e-5,
            'warmup_steps': 0,
            'model_name':'distilbert-base-uncased'
            }

huggingface_estimator = HuggingFace(entry_point='train_sagemaker.py',
                        source_dir='./scripts',
                        instance_type='ml.p3.2xlarge',
                        instance_count=1,
                        role=role,
                        transformers_version='4.26',
                        pytorch_version='1.13',
                        py_version='py39',
                        hyperparameters = hyperparameters)

# starting the train job with our uploaded datasets as input
huggingface_estimator.fit({'train': training_input_path, 'test': validation_input_path})
```

You can monitor the status of your training job in the AWS Sagemaker console.




We run the training job on a **ml.p3.2xlarge** instance. While the training job is running, check out the prices for the ml.p3.2xlarge and other instances on the AWS Sagemaker website and browse the Huggingface Sagemaker training documentation.

- https://aws.amazon.com/sagemaker/pricing/
- https://huggingface.co/docs/sagemaker/train

After your training job has successfully completed, you should see the message

```
Completed - Training job completed
```

You will also get a report on the training time and billable seconds. Calculate how much this training just cost you.

If you see an error, start debugging what went wrong and retry.

## 6. Deploy model for inference on AWS

Next, we deploy the model to a Sagemaker endpoint for inference and execute a query against it. Great success!

```
predictor = huggingface_estimator.deploy(1, "ml.g4dn.xlarge")


sentiment_input= {"inputs":" great movie"}

predictor.predict(sentiment_input)
```

## 6. Test model

Let's use our new endpoint to get predictions on the test set and compute the accuracy of the predictions.

```
def map_labels(label):
   mapping = {'LABEL_0': 0, 'LABEL_1': 1}
   return mapping[label]

sentiment_input= {"inputs": test_dataset["text"]}
test_output = predictor.predict(sentiment_input)
test_predictions = [map_labels(item['label']) for item in test_output]

# compute accuracy on test set
from sklearn.metrics import accuracy_score
accuracy_score(test_dataset['labels'], test_predictions)
```

### 10. Cleanup

**Please clean up all resources.**

- Stop any Sagemaker Studio JupyterLab instances, training jobs, and deployed endpoints

**Shut down AWS Sagemaker resources**
https://docs.aws.amazon.com/sagemaker/latest/dg/notebooks-run-and-manage-shut-down.html

## Conclusion

This session provided the first experience with how to train models with AWS Sagemaker using batch jobs instead of interactive notebook environments.

### Additional Resources

- Huggingface Sagemaker documentation incl. 1-hour recorded webinar
  https://huggingface.co/docs/sagemaker/getting-started

# Session 6: Amazon Bedrock

## Objective

In this session, you will experience AWS Bedrock, a new generative AI platform on AWS.

## Amazon Bedrock

Amazon Bedrock is a fully managed service offering access to foundation models from Amazon, other AI companies (Anthropic, Cohere, Meta) and open-source models. Unlike in AWS Sagemaker, you do not have to manage the deployment of models yourself but can directly consume them via APIs, similar to how you use OpenAI services.


## 1. Start Jupyter notebook

Start a JupyterLab space with a **ml.t3.medium instance** on AWS Sagemaker Studio.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

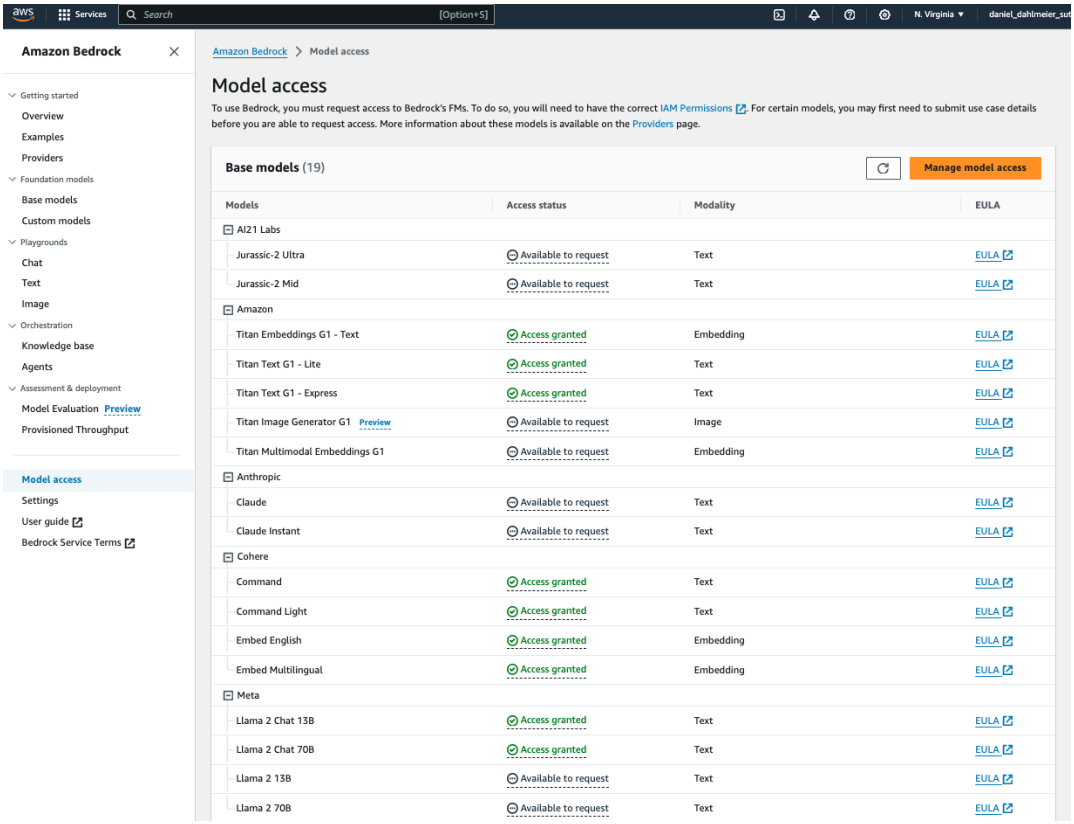
```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **06_bedrock.ipynb** in the folder sutd-mlops-course-code.

## 3. Manage model access on Amazon Bedrock

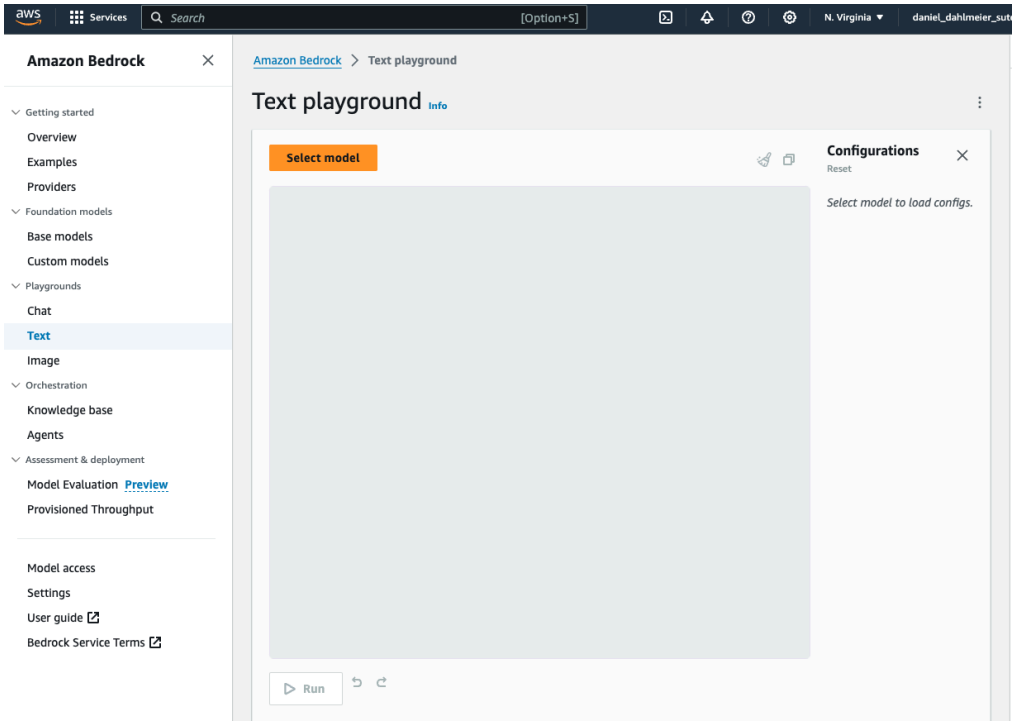
Before you can use models on Bedrock, you need to request access. For this lab, we have already requested access to the following models for you

- Amazon Titan G1 - Lite
- Amazon Titan G1 - Express
- Cohere Command
- Cohere Command Lite
- Llama 2 Chat 13B
- Llama 2 Chat 70B



## 4. Explore the Text Playground

Open the Text playground in the Bedrock console. The playground gives you a simple interface to interact with the models. Select a model and play with a few prompts. Compare the output of different models, for example different vendors and sizes. Also compared to ChatGPT (ChatGPT is not available in Bedrock, you need to open it in a separate browser window).




## 5. Connect to Amazon Bedrock service

Back to the notebook. Let's call models on Bedrock programmatically.

Run the first notebook cell to initialize the boto3 client. Boto3 is the AWS SDK for Python (no idea why it is called boto3). You can find the documentation here: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

Now you are ready to query Amazon Bedrock. Start by listing the available foundation models on Bedrock.

```
bedrock.list_foundation_models()
```

That’s a lot of models!

## 6. Prompt Foundation Models

Define a prompt and send it to Bedrock. Let’s start with the Amazon Titan Express model and ask it to write a poem about ML in the style of Shakespeare.

You need to specify the model you want to query. You can find code examples for each on the Bedrock console. Here is a video tutorial showing how to do it: https://www.youtube.com/watch?v=ab1mbj0acDo

```
bedrock.list_foundation_models()

# If you'd like to try your own prompt, edit this parameter!
prompt_data = """Command: Write me a poem about machine learning in the style of Shakespeare.

Poem:
"""

try:

    body = json.dumps({"inputText": prompt_data})
    modelId = "amazon.titan-text-express-v1"
    accept = "application/json"
    contentType = "application/json"

    response = bedrock_runtime.invoke_model(
        body=body, modelId=modelId, accept=accept, contentType=contentType
    )
    response_body = json.loads(response.get("body").read())

    print(response_body.get("results")[0].get("outputText"))

except botocore.exceptions.ClientError as error:
```

```
if error.response['Error']['Code'] == 'AccessDeniedException':
      print(f"\x1b[41m{error.response['Error']['Message']}\
          \nTo troubeshoot this issue please refer to the following resources.\
           \nhttps://docs.aws.amazon.com/IAM/latest/UserGuide/troubleshoot_access-denied.html\
           \nhttps://docs.aws.amazon.com/bedrock/latest/userguide/security-iam.html\x1b[0m\n")

else:
   raise error
```

Here is the results. I let someone else judge if this is worthy of Shakespeare but it looks like a poem and the topic is on point, too.

> Machine learning, a marvel of the age,
> A realm of data, where insights are made,
> Algorithms and models, in perfect harmony,
> Unveiling patterns, one by one.
>
> Data, the foundation, a treasure trove,
> Unveiled through analysis, a story to be told,
> Predictions and decisions, based on the rules,
> Machine learning, a powerful tool.
>
> Neural networks, with their intricate designs,
> Process vast amounts of data, with ease and glee,
> Learning and adapting, with each passing day,
> Machine learning, a force to be reckoned

Now, try out different prompts and models.

This concludes the Bedrock part of the session.

## Session 7: Retrieval Augmented Generation (RAG) with LangChain and AWS Sagemaker

Retrieval Augmented Generation, or RAG for short, is an emerging architectural pattern for building LLM applications. It addresses the important need to ground model answers in factual data and allow model responses to take into account updated data.

Imagine you have a model trained on data before 2021. If it is asked “Who is the president of the United States?” it would probably name Donald Trump or another US president which is present in its training data but not the current president, Joe Biden.


### 1. Open Notebook

Start a JupyterLab space with a **ml.t3.medium instance** on AWS Sagemaker Studio.

### 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **07_rag_sagemaker_langchain.ipynb**.

## 3. Development environment

Run the first two notebook cells to install the pypdf package and to load required packages.

## 4. Download data

The RAG application should be able to answer questions based on ingested documents. As example data, we will download a couple of PDF files containing the Letter to Shareholders by the CEO of Amazon.

```
# fetching data
!mkdir -p ./data

from urllib.request import urlretrieve
urls = [
  'https://s2.q4cdn.com/299287126/files/doc_financials/2023/ar/2022-Shareholder-Letter.pdf',
  'https://s2.q4cdn.com/299287126/files/doc_financials/2022/ar/2021-Shareholder-Letter.pdf',

'https://s2.q4cdn.com/299287126/files/doc_financials/2021/ar/Amazon-2020-Shareholder-Letter-and-1997-Shareholder-Letter.pdf',
  'https://s2.q4cdn.com/299287126/files/doc_financials/2020/ar/2019-Shareholder-Letter.pdf'
]

filenames = [
  'AMZN-2022-Shareholder-Letter.pdf',
  'AMZN-2021-Shareholder-Letter.pdf',
  'AMZN-2020-Shareholder-Letter.pdf',
  'AMZN-2019-Shareholder-Letter.pdf'
]
```

```
metadata = [
  dict(year=2022, source=filenames[0]),
  dict(year=2021, source=filenames[1]),
  dict(year=2020, source=filenames[2]),
  dict(year=2019, source=filenames[3])]

data_root = "./data/"

for idx, url in enumerate(urls):
  file_path = data_root + filenames[idx]
  urlretrieve(url, file_path)
```

We do some minimal pre-processing and remove the last pages which contain a copy of the first letter to shareholders from 1997.

## 5. Split documents

For the retrieval, i.e. search, part of RAG, we want to have smaller chunks of the documents which are hopefully providing succinct, relevant context to the question. We use LangChain to automatically split the documents into smaller text chunks. We use an overlap between chunks to make sure we do not lose relevant context.

**LangChain** is a popular framework for developing applications with LLMs. It provides abstractions over the various models and model services and provides implementations of common design patterns, like chain and agents.


Here is a video introduction to LangChain: https://www.youtube.com/watch?v=aywZrzNaKjs

```
# - in our testing Character split works better with this PDF data set
text_splitter = RecursiveCharacterTextSplitter(
  # Set a really small chunk size, just to show.
  chunk_size = 512,
  chunk_overlap  = 100,
)

docs = text_splitter.split_documents(documents)

print(f'# of Document Pages {len(documents)}')
print(f'# of Document Chunks: {len(docs)}')
```

## 6. Deploy embedding model endpoint

To allow our RAG application to find semantically relevant document chunks which do not have to rely on exact string matches, we compute text embeddings for each chunk. We deploy a text embedding model **all-MiniLM-L6-vs** from Hugging Face via Sagemaker Jumpstart.

Check out the model card on Hugging Face:
https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

Sagemaker Jumpstart provides a model hub where you can quickly provision foundation models for experimentation and fine tuning, as well as pre-build solutions. It is similar to Bedrock but you have more control over the models and need to deploy model endpoints yourself.

Here is a video walk through of Sagemaker Jumpstart:
https://www.youtube.com/watch?v=i4W7SfP6_38

(If by this time, you feel that AWS has a lot of similar services with overlapping functionality, you are completely right!)

```
from sagemaker.jumpstart.model import JumpStartModel

# default instance is ml.g5.2xlarge
embedding_model_id, embedding_model_version = "huggingface-textembedding-all-MiniLM-L6-v2", "*"
model = JumpStartModel(model_id=embedding_model_id, model_version=embedding_model_version)
embedding_predictor = model.deploy()
```

## 7. Compute document embeddings

Now we can compute the embeddings for each document chunk by calling the embedding endpoint.

```
from typing import Dict, List
from langchain.embeddings import SagemakerEndpointEmbeddings
from langchain.embeddings.sagemaker_endpoint import EmbeddingsContentHandler
import json


class CustomEmbeddingsContentHandler(EmbeddingsContentHandler):
  content_type = "application/json"
  accepts = "application/json"

  def transform_input(self, inputs: list[str], model_kwargs: Dict) -> bytes:
    input_str = json.dumps({"text_inputs": inputs, **model_kwargs})
    return input_str.encode("utf-8")

  def transform_output(self, output: bytes) -> List[List[float]]:
    response_json = json.loads(output.read().decode("utf-8"))
    return response_json["embedding"]

embeddings_content_handler = CustomEmbeddingsContentHandler()

embeddings = SagemakerEndpointEmbeddings(
```

```
  endpoint_name=embedding_model_endpoint_name,
  region_name=aws_region,
  content_handler=embeddings_content_handler,
)
```

Now,we put all document chunks and embeddings into a vector store index for fast lookup. There are many options for vector stores and vector databases out there. We are using the Facebook AI Similarity Search (Faiss) library which provides fast, approximate nearest neighbor search and comes integrated with LangChain.

```
from langchain.schema import Document
from langchain.vectorstores import FAISS

db = FAISS.from_documents(docs, embeddings)
```

## 8. Deploy LLM endpoint for generation

Now let's put the "G" into RAG. To generate the final answer based on the returned document context, we need an LLM model and a prompt template which puts the question and context together with some instruction. We deploy a Llama 2 7B chat model endpoint using Sagemaker Jumpstart.

```
llm_model_id, llm_model_version = "meta-textgeneration-llama-2-7b-f", "2.*"
llm_model = JumpStartModel(model_id=llm_model_id, model_version=llm_model_version)
llm_predictor = llm_model.deploy()

llm = SagemakerEndpoint(
      endpoint_name=llm_model_endpoint_name,
      region_name=aws_region,
      model_kwargs={"max_new_tokens": 1000, "top_p": 0.9, "temperature": 1e-11},
      endpoint_kwargs={"CustomAttributes": 'accept_eula=true'},
      content_handler=qa_content_handler
  )
```

Llama 2 is a family of powerful LLMs built and released by Meta. Here is the website: https://ai.meta.com/llama/

## 9. Query the LLM *without* RAG

You can invoke this LLM object directly to get a baseline response without any contextual information provided. You'll notice the answer to the question 'How has AWS evolved?' is more about what AWS has done rather than a more internal take on how AWS has evolved. This is likely due to the corpus of data that the LLM was trained on which contained a large amount of articles from the internet.

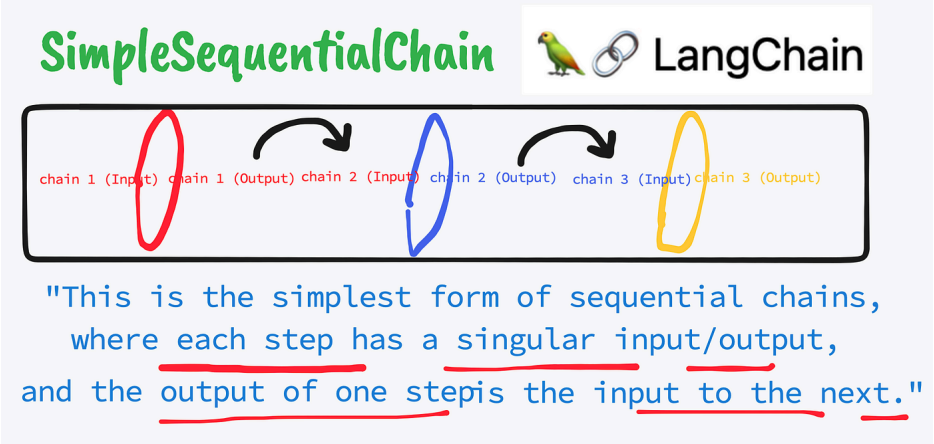
Note that this is not a bad response by any stretch, but it might not be the response you're looking for.

You'll see how context can evolve the response in a moment.

```
query = "How has AWS evolved?"
llm.predict(query)
```

## 10. Query the RAG application

With the LLM endpoint object created, you are ready to create your first chain!
Chains are a concept to link together multiple LLM calls or processing steps by providing the result of one step as input to the next. This blog explains the idea behind chains in LangChain with code examples.




In our example we will provide the result of the retrieval step as input to the generation step. This chain is a simple example using LangChain's RetrievalQA chain, which will:

1. Take a query as input
2. Generate query embeddings
3. Query the vector database for relevant document chunks based on the query embedding
4. Inject the context and original query into the prompt template
5. Invoke the LLM with the completed prompt
6. Return the LLM result

For more examples how Chains can be used in applications, check out this blog: https://shishirsingh66g.medium.com/langchain-concepts-part-6-chain-37ae677fec3b

```
qa_chain = RetrievalQA.from_chain_type(
  llm,
  chain_type='stuff',
  retriever=db.as_retriever(),
  return_source_documents=True,
  chain_type_kwargs={"prompt": PROMPT}
)

query = "How has AWS evolved?"
result = qa_chain({"query": query})
print(f'Query: {result["query"]}\n')
```

```
print(f'Result: {result["result"]}\n')
print(f'Context Documents: ')
for srcdoc in result["source_documents"]:
    print(f'{srcdoc}\n')
```

Great success! Try out a few more examples.

### 11. Cleanup

**Please clean up all resources.**

- Stop any Sagemaker Studio JupyterLab instances, training jobs, and deployed endpoints
- Shut down the kernel and logout of the SUTD Education Cluster
- Check the AWS Cost Explorer to review your AWS spend

**Shut down AWS Sagemaker resources**
https://docs.aws.amazon.com/sagemaker/latest/dg/notebooks-run-and-manage-shut-down.html

## Conclusion

In this session, you learned how to use LangChain and AWS Bedrock to build a question-answering RAG system, which can answer questions over your data.

# Session 8: Finetune Llama models

## Objective

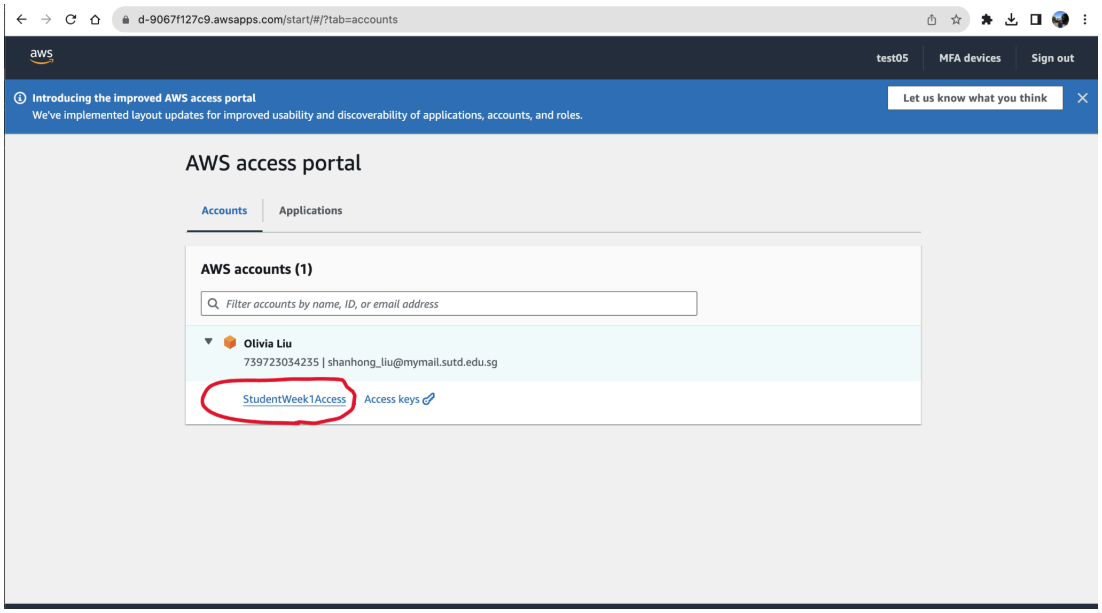
In this session, you will finetune an LLM model. Finetuning allows to specialize an LLM model on a specific task, change the behavior of a model and to align LLMs with human instructions.

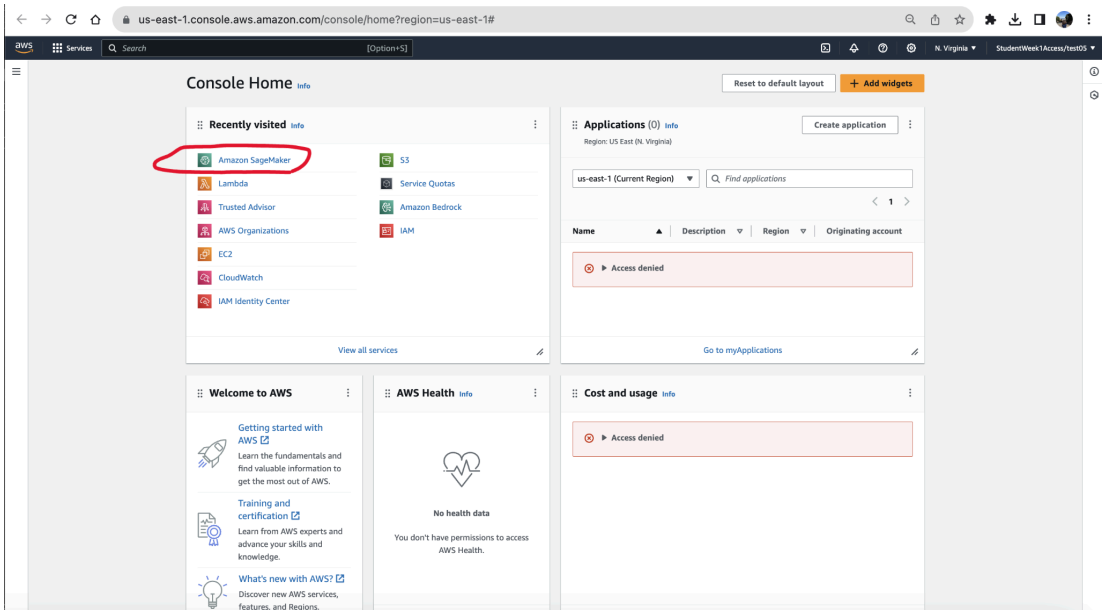
Similar to the BERT finetuning exercise, we will experiment with different methods: running the finetuning job directly from the notebook environment, running a training job on AWS Sagemaker and using Sagemaker Jumpstart.

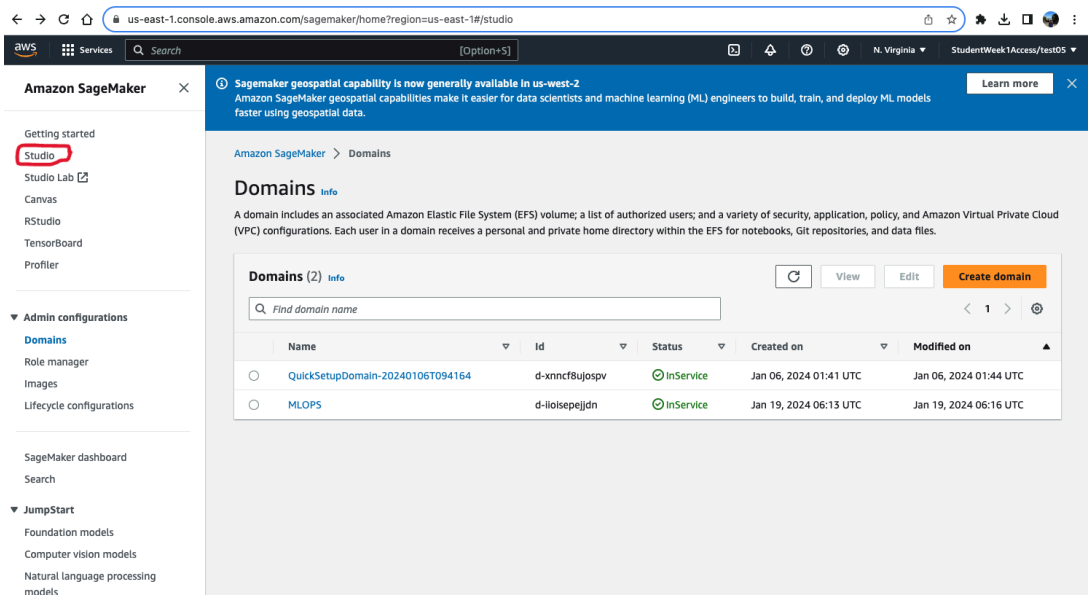
## Understanding Llama 2 and OpenLLaMA

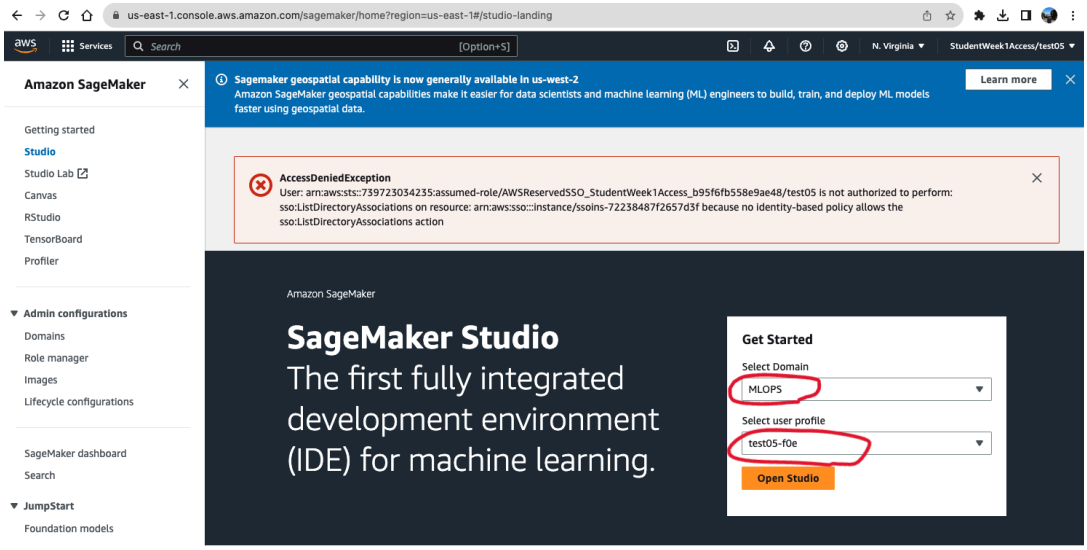
We will finetune Llama 2 7B, a popular open-source LLM from Meta and OpenLLaMA 3B, a smaller reproduction of LLamA. For background on Llama 2 and OpenLLaMA, refer to the paper and github repo below.

- Llama 2 paper
- OpenLLamA github




us-east-1.console.aws.amazon.com/console/home?region=us-east-1#
Console Home Info
Reset to default layout
+ Add widgets
Recently visited Info
Amazon SageMaker
Lambda
Trusted Advisor
AWS Organizations
EC2
CloudWatch
IAM Identity Center
S3
Service Quotas
Amazon Bedrock
IAM
View all services
Applications (0) Info
Region: US East (N. Virginia)
Create application
us-east-1 (Current Region)
Find applications
Name
Description
Region
Originating account
Access denied
Go to myApplications
Welcome to AWS
Getting started with AWS
Learn the fundamentals and find valuable information to get the most out of AWS.
Training and certification
Learn from AWS experts and advance your skills and knowledge.
What's new with AWS?
Discover new AWS services, features, and Regions.
AWS Health Info
No health data
You don't have permissions to access AWS Health.
Cost and usage Info
Access denied


us-east-1.console.aws.amazon.com/sagemaker/home?region=us-east-1#/studio
Amazon SageMaker
Getting started
Studio
Studio Lab
Canvas
RStudio
TensorBoard
Profiler
Admin configurations
Domains
Role manager
Images
Lifecycle configurations
SageMaker dashboard
Search
JumpStart
Foundation models
Computer vision models
Natural language processing models
Sagemaker geospatial capability is now generally available in us-west-2
Amazon SageMaker geospatial capabilities make it easier for data scientists and machine learning (ML) engineers to build, train, and deploy ML models faster using geospatial data.
Learn more
Amazon SageMaker > Domains
Domains Info
A domain includes an associated Amazon Elastic File System (EFS) volume; a list of authorized users; and a variety of security, application, policy, and Amazon Virtual Private Cloud (VPC) configurations. Each user in a domain receives a personal and private home directory within the EFS for notebooks, Git repositories, and data files.
Domains (2) Info
View
Edit
Create domain
Find domain name
Name
Id
Status
Created on
Modified on
QuickSetupDomain-20240106T094164
d-xnncf8ujospv
InService
Jan 06, 2024 01:41 UTC
Jan 06, 2024 01:44 UTC
MLOPS
d-iioisepejjdn
InService
Jan 19, 2024 06:13 UTC
Jan 19, 2024 06:16 UTC




## Finetuning in notebook

## 1. Start Jupyter notebook

Start a JupyterLab space on AWS Sagemaker Studio.

> **Note**: make sure the instance provides the GPU computation. For this exercise, you are recommended to use an **ml.g5.4xlarge** or **ml.g4dn.4xlarge** instance with **20GB storage**.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **08_finetune_llama_notebook.ipynb** in the folder sutd-mlops-course-code.

### 3. Development environment

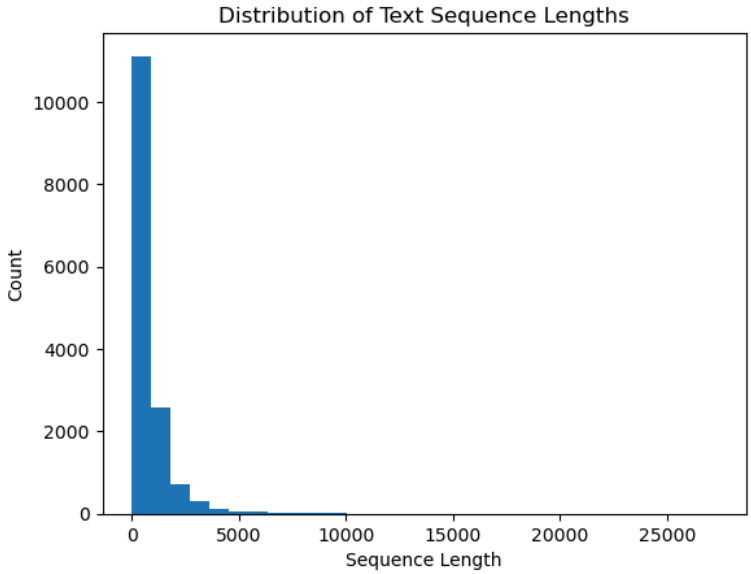
Run the first cells to install all required dependencies and import the necessary packages. All this should feel very familiar by now.

### 4. Data Preparation

Load the databricks-dolly-15k dataset. The Dolly dataset is an open-source instruction-following dataset created by Databricks. The dataset contains pairs of human-generated instructions, an optional context, and responses for various categories of tasks.Read about the Dolly dataset and the Dolly LLM created with this dataset on the Databricks blog.

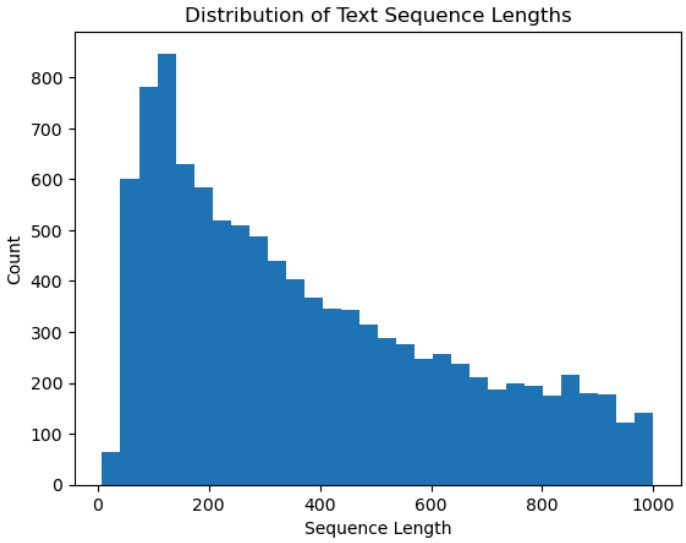
```
from datasets import load_dataset

dbricks_15k_dataset_base = load_dataset("databricks/databricks-dolly-15k")
```

We plot the length of the instances and see that it has a long-tail with the majority of instances being below 2000 characters and a few very long sequences with more than 25k characters.



For GPU training, it is more efficient if your training instances are of roughly the same length.

We drop very long instances and split the dataset into a training and test set. The distribution of the reduced training set looks more well behaved.




We use a prompt template to put the instruction, context and response into the appropriate single “text” field.

```
{   'category': 'brainstorming',
    'context': '',
    'instruction': 'Provide 5 different ways you could improve a typical '
                   "home's ambience",
    'response': '1.\tRemove clutter from common spaces \n'
                '2.\tAdd artwork to blank walls \n'
                '3.\tUse common color themes within rooms  \n'
                '4.\tMatch lighting to room purpose and vibe  \n'
                '5.\tPlace a houseplant near the edge of the living room',
    'text': 'Below is an instruction that describes a task. Write a response '
            'that appropriately completes the request.\n'
            '\n'
            '### Instruction:\n'
            "Provide 5 different ways you could improve a typical home's "
            'ambience\n'
            '\n'
            '### Response:\n'
            '1.\tRemove clutter from common spaces \n'
            '2.\tAdd artwork to blank walls \n'
            '3.\tUse common color themes within rooms  \n'
```

```
        '4.\tMatch lighting to room purpose and vibe  \n'
        '5.\tPlace a houseplant near the edge of the living room'}
```

## 5. Prepare Training

Even though we are using “small” LLM models, finetuning them requires large amounts for GPU VRAM. Parameter efficient finetuning (PEFT) methods allow more efficient (and cheaper) model finetuning by just updating a small set of parameters instead of the entire model We are using the popular QLoRA method which adds a quantized low-rank decomposition of the model parameters and only updates these parameters while keeping the rest of the model fixed.

Check out the QLoRA paper and the below blog article to learn about QLoRA.

- QLoRA paper
- Huggingface Blog Making LLMs even more accessible with bitsandbytes, 4-bit quantization and QLoRA

As the base model, we are using the 3 billion parameter model "openlm-research/open_llama_3b_v2" from Berkely.

Instead of implementing QLoRA directly in Pytorch or another deep learning framework, we are using the supervised finetuning SFTT API which gives us a more high-level abstraction. There are many parameters which can be set to configure the training job. Check out this blog which describes the parameters in great detail

- Notes on fine-tuning Llama 2 using QLoRA: A detailed breakdown. Blog by Ogban Ugot

```
# Load LoRA configuration
peft_config = LoraConfig(
    lora_alpha=lora_alpha,
    lora_dropout=lora_dropout,
    r=lora_r,
    bias="none",
    task_type="CAUSAL_LM",
)

# Set training parameters
training_arguments = TrainingArguments(
    output_dir=output_dir,
    num_train_epochs=num_train_epochs,
    per_device_train_batch_size=per_device_train_batch_size,
    gradient_accumulation_steps=gradient_accumulation_steps,
```

```
    optim=optim,
    save_steps=save_steps,
    logging_steps=logging_steps,
    learning_rate=learning_rate,
    weight_decay=weight_decay,
    fp16=fp16,
    bf16=bf16,
    max_grad_norm=max_grad_norm,
    max_steps=max_steps,
    warmup_ratio=warmup_ratio,
    group_by_length=group_by_length,
    lr_scheduler_type=lr_scheduler_type,
    report_to="wandb"
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=formatted_dataset["train"],
    eval_dataset=formatted_dataset["test"],
    peft_config=peft_config,
    dataset_text_field="text",
    max_seq_length=max_seq_length,
    tokenizer=tokenizer,
    args=training_arguments,
    packing=packing,
)
```

## 6. Train model

Now we are ready to start training the model by calling trainer.train(). This is where the training happens but all the parameters have already been configured before, so this single command looks almost boring.

The training gets logged to Weights and Biases. You will be prompted to enter your API key. After that, follow the training run on the command line and on Weights and Biases.

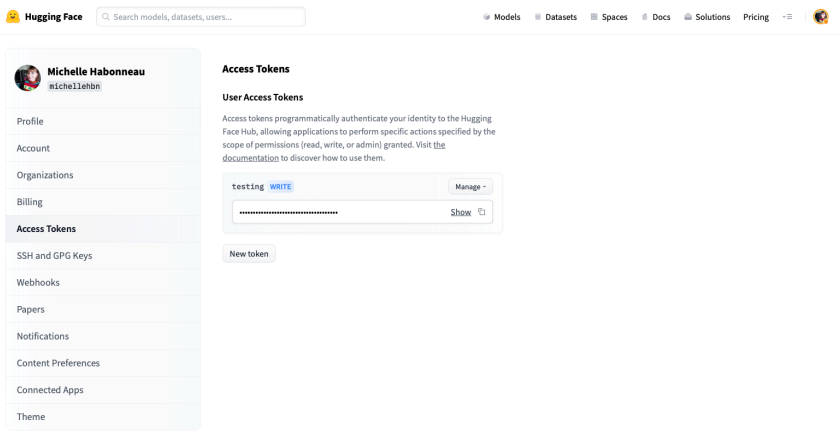
```
# Train model
trainer.train()

# Save trained model
trainer.model.save_pretrained(new_model)
```

## 7. Test model

To test the mode, we run the evaluation on the test set and print the resulting metrics.

## 8. Upload model to Huggingface

After training has successfully concluded, we upload the learned parameter weights to the Huggingface model hub. To allow your code to push artifacts to Huggingface, you need to provide a [Huggingface User Access token](#).




## 9. Run inference

Let's see how the model performs. Load the model back and merge it with the LoRA weights. Make a few inference calls with instructions and context (optional) and see how the model performs.

```
make_inference("Identify the odd one out and explain your choice.", "Orange, Green, Airplane.")
```

## 10. Cleanup

**Please clean up all resources.**

- Stop your Sagemaker Studio JupyterLab instances.

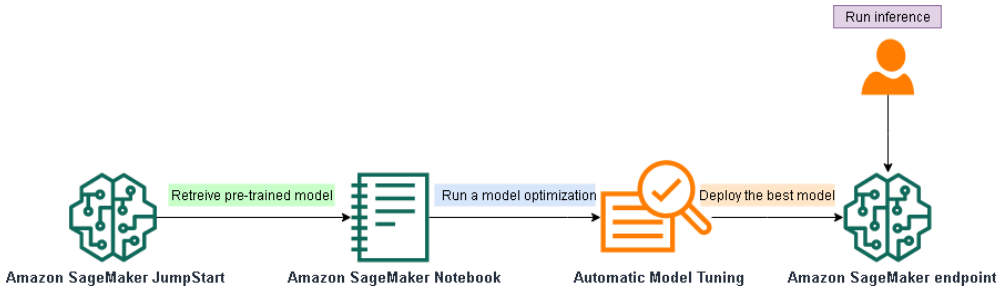
**Shut down AWS Sagemaker resources**

https://docs.aws.amazon.com/sagemaker/latest/dg/notebooks-run-and-manage-shut-down.html

This concludes the notebook finetuning part. Next, we will accomplish model finetuning with the help of Sagemaker Jumpstart.

## Finetuning with Sagemaker Jumpstart

Amazon Jumpstart is a service to deploy, finetune and evaluate LLM models. It provides higher-level APIs to work with LLMs compared to working with the models directly.




## 1. Start Jupyter notebook

Start a JupyterLab space on AWS Sagemaker Studio.

> **Note**: We will use GPU computation via Sagemaker Jumpstart and do not need to have a local GPU on the JupyterLab space. For this exercise, you are recommended to use an **ml.t3.medium** instance.

## 2. Checkout Github repository

Check out the Github repo by opening a terminal and run

```
~$ git clone https://github.com/ddahlmeier/sutd-mlops-course-code.git
```

Open the notebook **08_finetune_llama_sagemaker_jumpstart.ipynb** in the folder sutd-mlops-course-code.

## 3. Development environment

Run the first cells to install the dataset library with S3 support.

## 4. Deploy Llama 2 model endpoint

Before we dive into finetuning, we deploy a pre-trained **Llama 2 7B text generation model** for inference. We perform an inference call, asking about the meaning of life, just to ensure the endpoint is working.

```
model_id, model_version = "meta-textgeneration-llama-2-7b", "3.1.0*"

from sagemaker.jumpstart.model import JumpStartModel

pretrained_model = JumpStartModel(model_id=model_id, model_version=model_version)
pretrained_predictor = pretrained_model.deploy(accep_eula=True)
```

## 5. Data Preparation

Again, we are using the databricks-dolly-15k dataset but this time, we are only using the closed question-answering examples from the dataset. As before, we filter out too long instances.

```
from datasets import load_dataset
```

```
# load training data set
dolly_dataset = load_dataset("databricks/databricks-dolly-15k", split="train")

# Filter for question answering examples
qa_dataset = dolly_dataset.filter(lambda example: example["category"] == "closed_qa")
qa_dataset = qa_dataset.remove_columns("category")
```

Then we upload the training dataset and the prompt template as json files to S3.

```
from datasets import load_dataset

# load training data set
dolly_dataset = load_dataset("databricks/databricks-dolly-15k", split="train")

# Filter for question answering examples
qa_dataset = dolly_dataset.filter(lambda example: example["category"] == "closed_qa")
qa_dataset = qa_dataset.remove_columns("category")

from sagemaker.s3 import S3Uploader
import sagemaker
import random

bucket = sagemaker.Session().default_bucket()
local_data_file = "training.jsonl"
train_data_location = f"s3://{bucket}/dolly_dataset"
S3Uploader.upload(local_data_file, train_data_location)
S3Uploader.upload("template.json", train_data_location)
print(f"Training data: {train_data_location}")
```

## 6. Train model

Now we execute the training using Sagemaker Jumpstart. Unlike the local training in the notebook which we did before, we are now running training remotely on the Sagemaker platform. We create an estimator class with the right parameters and trigger the training run.

Follow the training progress via the log messages.

```
from sagemaker.jumpstart.estimator import JumpStartEstimator
```

```
estimator = JumpStartEstimator(
    model_id=model_id,
    model_version=model_version,
    instance_type="ml.g5.12xlarge",
    instance_count=1,
    environment={"accept_eula": "true"}
)

# By default, instruction tuning is set to false
estimator.set_hyperparameters(instruction_tuned="True",
                          epoch="1",
                          max_input_length="1024")
estimator.fit({"training": train_data_location})
```

## 7. Deploy and test model

Once training is completed, we can deploy the model to an endpoint for inference with a simple function call.

```
finetuned_predictor = estimator.deploy()
```

Then we query the non-finetuned and the finetuned model endpoint with a few test questions. We show the responses from both models side by side.
What difference do you see? Has the finetuning improved the question-answering capabilities of the model?

How long did the training take and how much did it cost?

## 10. Cleanup

**Please clean up all resources.**

- Run the last cell to delete the models and endpoints.
- Stop any Sagemaker Studio JupyterLab instances, training jobs, and deployed endpoints
- Check the AWS Cost Explorer to review your AWS spend

**Shut down AWS Sagemaker resources**
https://docs.aws.amazon.com/sagemaker/latest/dg/notebooks-run-and-manage-shut-down.html

# Conclusion

In this session, you learned how to finetune LLM models using Huggingface in a notebook environment and via Sagemaker studio.

### Additional Resources

- DeepLearning.AI short course on Finetuning Large Language Models

# Appendix

Overview of required Service Quotas for complete class

| Service Quota | Required quota | Comment |
|---|---|---|
| ml.t3.medium | 30 (already fulfilled) | 1 per student + 2 instructors + buffer |
| ml.p3.2xlarge for training job usage | 30 (granted) | 1 per student + 2 instructors + buffer |
| ml.g4dn.xlarge for endpoint usage | 30 (granted) | 1 per student + 2 instructors + buffer |
| ml.g5.2xlarge for endpoint usage | 56 (granted) | 2 per student + 2 instructors + buffer |
| ml.g5.4xlarge for notebook instance usage | 30 (granted) | 1 per student + 2 instructors + buffer |
| ml.g5.12xlarge for training job usage | 30 (granted) | 1 per student + 2 instructors + buffer |

# Section 1: Traditional ML lifecycle

TODO: explain a bit what this section of the course is about which is going through the entire lifecyle of a machine learning model development.

- Inspect and label data set
- Train model
- Test model
- Deploy model for inference

## Session 1: Setup AWS environment

(This session could be done on SUTD Cluster or on AWS)
In this session, you will

- Create an AWS account
- Link your AWS account to the instructor account to allow consolidated billing
- Setup Sagemaker
- Request the necessary service quotas
- Overview about AWS
- Learn how to track your AWS costs
- SET CLEAR BUDGET PER STUDENT!
- Introduction to Sagemaker
- Create Jupyterlab, create space with default settings (ml.t3.medium)
- Short introduction how use jupyter notebooks and git
- Check out the course github repo
- Download movie review sentiment data set
- Explore and analyze movie review data with matplotlib
- Label a subset of the reviews yourself using AWS Sagemaker Ground Truth
- Compute the inter-annotator agreement between your annotations and the reference labels

Introduction to Sagemaker https://www.youtube.com/watch?v=agq-C4XyL3E

AWS tutorials getting started with Amazon Sagemaker:

https://aws.amazon.com/sagemaker/getting-started/

- Tutorial how to label data with ground truth
  https://aws.amazon.com/tutorials/machine-learning-tutorial-label-training-data/
- Build and train a machine learning model locally
  https://aws.amazon.com/tutorials/machine-learning-tutorial-build-model-locally/

  Daniel: I get a couple of errors with dependency issues and Int64 data types. I think this tutorial is not working out of the box. This needs to be tested again

| Service Quota | AWS default quota | Apply for this quota |
|---|---|---|
| Total domains | 2 | N/A |
| ml.g5.xlarge for notebook instance usage | 0 | 1 |
| ml.g5.xlarge for cluster usage | 0 | 1 |
| ml.g5.xlarge for training job usage | 0 | 1 |
| ml.g5.xlarge for endpoint job usage | 0 | 2 |
| ml.g5.2xlarge for notebook instance usage | 0 | 1 |
| ml.g5.2xlarge for cluster usage | 0 | 1 |
| ml.g5.2xlarge for training job usage | 0 | 1 |
| ml.g5.2xlarge for endpoint job usage | 0 | 2 |
| TODO check other quotas and add here | | |

TODO: Check in detail for each notebook what instances are required and add the required quotas here

## Session 2: Train linear classifier

(This session could be done on SUTD Cluster or on AWS)

In this session, you will

- Train a linear classifier on a sentiment classification task
- Log the experiment details using wandb
- Experiment with different hyperparamters

TODO: this seems a bit too short?

Wandb scikit tutorial
https://wandb.ai/lavanyashukla/visualize-sklearn/reports/Visualize-Scikit-Learn-Models-with-Weights-Biases--Vmlldzo0ODIzNg

## Session 3: Finetune BERT classifier

(This session could be done on SUTD Cluster or on AWS)

- Train Distillbert classifier for the sentiment task
- TODO: check what AWS instance is required for Distillbert and BERT. For Distillbert ml.c5.9xlarge seems to be enough (https://github.com/data-science-on-aws/data-science-on-aws/blob/oreilly-book/07_train/02_Train_Reviews_BERT_Transformers_TensorFlow_ScriptMode.ipynb). Here they are using p3.2x. Large for BERT training and ml.g4dn.xlarge for inference. ml.c5.2xlarge should work for training too
- Minimum instance to finetune distillbert is ml.m5.large instance has 1 GPU, 4 vCPUs and 16 GB of memory which costs $0.144 per Hour.
- Deploy BERT models via Sagemaker endpoints deployment
- Do inference calls
- Compare accuracy and costs for linear model and Distillbert/BERT
-

Tutorial finetune Distillbert
https://github.com/aws-samples/finetune-deploy-bert-with-amazon-sagemaker-for-hugging-face/blob/main/finetune-distilbert.ipynb
Another one from hugginface
https://github.com/huggingface/notebooks/blob/main/sagemaker/01_getting_started_pytorch/sagemaker-notebook.ipynb

Wandb tutorial BERT and Distillbert
https://wandb.ai/jack-morris/david-vs-goliath/reports/Does-model-size-matter%3F-A-comparison-of-BERT-and-DistilBERT--VmlldzoxMDUxNzU?_gl=1%2Ac7rw1q%2A_ga%2AMjI5NjcxNjIuMTcwMzkwOTI0MA..%2A_ga_JH1SJHJQXJ%2AMTcwNDI5MTgyNi4xMi4xLjE3MDQyOTE4OTcuNjAuMC4w
https://wandb.ai/wandb_fc/articles/reports/Sentence-Classification-with-HuggingFace-BERT-and-Hyperparameter-Optimization-with-W-B--Vmlldzo1NDI2ODg1

# Section 2: LLMs

Task: build chatbot about SUTD

## Session 4: AWS Bedrock and RAG QA-bot

(This session has to be done on AWS)

- Deploy LLM models on AWS Bedrock
- Experiment with prompt engineering

1. Prompt engineering with public models on Bedrock
    a. Simple deploying and prompting of model https://github.com/generative-ai-on-aws/generative-ai-on-aws/blob/main/12_bedrock/01_bedrock_overview.ipynb
2. RAG with additional documents about SUTD eg PDF
    a. Use AWS Sagemaker and langchain example from above
    b. Pinecone is very easy to use and free for a start too but somehow super slow to create the embeddings (openAI ada)
3. Evaluate RAG QA model performance (correctness, groundedness, relevance of contex)

RAG example on AWS Sagemaker with langchain. This seems to work pretty well, I just need to request more GPU instance quota. Let us use this one
https://github.com/generative-ai-on-aws/generative-ai-on-aws/blob/main/09_rag/01_langchain_llama2_sagemaker.ipynb
https://www.youtube.com/watch?v=J_tCD_J6w3s

TODO: create a RAG example without AWS Sagemaker so that it could be run on SUTD cluster if necessary

## Session 5: LLM finetuning

(This session has to be done on AWS)

-Finetuning small LLM model eg Pythia 70M, 401M models with continued pre-training and instruction tuning

- Try instruction tuning for Mistral 7B or LLama 7B model (at least on a small data set for one epoch
- Compare non-finetuned and finetuned models

TODO: add uploading model to HF

## Any more hands-on sessions?

Other topics from course proposal are

- ML testing
- Pipelining and service
- Logging and monitoring

Rest of the hands-on sessions are free to work on assignments

# Resources

Sagemaker documentation: https://docs.aws.amazon.com/sagemaker/

Finetuning Mistral 7B with Sagemaker jumpstart using the web console and all defaults works. The resulting model is not great but it can be deployed to an endpoint and responds to requests

Another mistral 7B tutorial but they also use bits and bites which throws an error on sagemaker
https://www.youtube.com/watch?v=6DGYj1EEWOw&list=PLxqBkZuBynVTzqUQCQFgetR97y1X_1uCI&index=14

Some tips how to install bites and bites on sagemaker notebook

https://aws.amazon.com/blogs/machine-learning/interactively-fine-tune-falcon-40b-and-other-llms-on-amazon-sagemaker-studio-notebooks-using-qlora/

This deeplearning.ai video an colab are pretty nice and clean examples for finetuning openLlama
https://www.youtube.com/watch?v=eTieetk2dSw
https://github.com/FourthBrain/Building-with-Instruction-Tuned-LLMs-A-Step-by-Step-Guide?tab=readme-ov-file

TinyLlama finetuning example
https://colab.research.google.com/drive/1AZghoNBQaMDgWJpi4RbffGM1h6raLUj9?usp=sharing#scrollTo=ekOmTR1hSNcr