

## ДОДАТОК А

Вебдодаток Flask:

app.py file:

```
from flask import Flask, render_template, jsonify, request, flash, redirect
from werkzeug.datastructures import FileStorage
from werkzeug.utils import secure_filename
from fileinput import filename
import config
import model_api
import os
```

```
def page_not_found(e):
    return render_template("404.html"), 404
```

```
ALLOWED_EXTENSIONS = set(["pdf", "txt", "docx"])
```

```
def allowed_file(filename):
    return "." in filename and filename.rsplit(".", 1)[1].lower() in
ALLOWED_EXTENSIONS
```

```
app = Flask(__name__)
app.secret_key = "31242"
```

```
path = os.getcwd()
# file Upload
UPLOAD_FOLDER = os.path.join(path, "data")
if not os.path.isdir(UPLOAD_FOLDER):
    os.mkdir(UPLOAD_FOLDER)
app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER
```

```
app.register_error_handler(404, page_not_found)
```

```
@app.route("/upload", methods=["POST"])
```

```
def upload_file():
    if request.method == "POST":
        if "file" not in request.files:
            flash("No file part")
            return redirect(request.url)
```

```
        context_file = request.files["file"]
```

```
        if context_file.filename == "":
```

```

    flash("No file selected for uploading")
    return redirect(request.url)

    if context_file and allowed_file(context_file.filename):
        filename = secure_filename(context_file.filename)
        context_file.save(os.path.join(app.config["UPLOAD_FOLDER"],
filename))
        flash("File successfully uploaded")
        return redirect("/")
    else:
        flash("Allowed file types are txt, pdf, png, jpg, jpeg, gif")
        return redirect(request.url)

@app.route("/", methods=["POST", "GET"])
def index():
    if request.method == "POST":
        question = request.form["question"]
        context = request.form["context"]

        res = {}
        res["answer"] = model_api.getModelAPIResponse(question, context)
        return jsonify(res), 200
    return render_template("index.html", **locals())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8888", debug=True)

```

Model API file:

```

import config
import requests
import json

# call the aws Gateway API to get the response
def getModelAPIResponse(question:str, context:str) -> str:
    url = "https://0er2zt8aoj.execute-api.us-east-1.amazonaws.com/"

    payload = json.dumps({
        "question": question,
        "context": context
    })
    headers = {
        'Content-Type': 'application/json'

```

```
}
```

```
response = requests.request("POST", url, headers=headers, data=payload)
```

```
return response.text
```

index.html file:

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
<head>
```

```
<meta charset="utf-8">
```

```
<meta http-equiv="x-ua-compatible" content="ie=edge">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

```
<meta name="description" content="BERT QAS">
```

```
<title>Question and Answering system</title>
```

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet"
```

```
integrity="sha384-GLh1TQ8iRABdZL16O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD" crossorigin="anonymous">
```

```
</head>
```

```
<body>
```

```
<header>
```

```
<!-- Fixed navbar -->
```

```
<nav class="navbar navbar-expand-md navbar-dark fixed-top bg-dark">
```

```
<div class="container-fluid">
```

```
<a class="navbar-brand" href="#">nogyxo</a>
```

```
<button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarCollapse"
```

```
aria-controls="navbarCollapse" aria-expanded="false" aria-label="Toggle navigation">
```

```
<span class="navbar-toggler-icon"></span>
```

```
</button>
```

```
<div class="collapse navbar-collapse" id="navbarCollapse">
```

```
<ul class="navbar-nav me-auto mb-2 mb-md-0">
```

```
</ul>
```

```
</div>
```

```
</div>
```

```
</nav>
```

```

</header>

<!-- Begin page content -->
<main class="flex-shrink-0">
  <div class="container">
    <br>
    <br>
    <h1 class="mt-5">The Question and Answering system</h1>
    <p class="lead">
      This project created to illustrate the power of BERT in question answering
      tasks.
    </p>

    <div id="list-group" class="list-group w-auto">
      <a href="#" class="list-group-item list-group-item-action d-flex gap-3 py-
3">
        
        <div class="d-flex gap-2 w-100 align-items-center flex-column">
          <div>
            <p class="mb-0 opacity-75">Provide me file with context</p>
          </div>
          <div>
            <form id="upload-file" method="POST" enctype="multipart/form-
data">
              <fieldset>
                <input name="file" type="file">
                <button id="upload-file-btn" type="button">Upload</button>
              </fieldset>
            </form>
          </div>
          <div>
            <p class="mb-0 opacity-75">Or insert the text in the input box</p>
          </div>
          <input type="text" class="form-control" id="context-input">
        </div>
      </a>
    </div>
    <div class="input-group mb-3">
      <input type="text" class="form-control" id="question-input">

```

```
    <div class="input-group-append">
      <button id="ask-button" class="btn btn-primary">Ask a
question</button>
    </div>
  </div>
</div>
</main>
```

```
<script src="https://code.jquery.com/jquery-3.6.3.min.js"
  integrity="sha256-
pvPw+upLPUjgMXY0G+8O0xUf+/Im1MZjXxxgOcBQBxU="
crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/js/bootstrap.bundle.min.js"
  integrity="sha384-
w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBT
kF7CXvN"
  crossorigin="anonymous"></script>
```

```
<script>

$("#ask-button").click(function () {
  var context = $("#context-input").val();
  var question = $("#question-input").val();
  let html_data = "";
  html_data += `
<a href="#" class="list-group-item list-group-item-action d-flex gap-3 py-
3">
  <div class="d-flex gap-2 w-100 justify-content-between">
    <div>
      <p class="mb-0 opacity-75">${question}</p>
    </div>
  </div>
</a>
`;
$("#question-input").val("");
$("#list-group").append(html_data);

//AJAX CALL TO SERVER
$.ajax({
  type: "POST",
  url: "/",
  data: { 'context': context, 'question': question },
```

```

    success: function (data) {
        let model_data = "";
        model_data += `
            <a href="#" class="list-group-item list-group-item-action d-flex gap-3
py-3">
                
                <div class="d-flex gap-2 w-100 justify-content-between">
                    <div>
                        <p class="mb-0 opacity-75">${data.answer}</p>
                    </div>
                </div>
            </a>
        `;
        $("#list-group").append(model_data);
    }
});

$(function () {
    $('#upload-file-btn').click(function () {
        var form_data = new FormData($('#upload-file')[0]);
        $.ajax({
            type: 'POST',
            url: '/upload',
            data: form_data,
            contentType: false,
            cache: false,
            processData: false,
            success: function (data) {
                console.log('Success!');
            },
        });
    });
});
</script>
</body>
</html>

```

Lambda file:

```
import os
import json
import boto3
```

```
# grab environment variables
```

```
ENDPOINT_NAME = os.environ['ENDPOINT_NAME']
```

```
def lambda_handler(event, context):
```

```
    if not event:
```

```
        return {
```

```
            'statusCode': 200,
```

```
            'headers': {'Content-Type': 'application/json'},
```

```
            'body': json.dumps({'Event': event, 'message': 'The event is empty'})
```

```
        }
```

```
    if 'requestContext' in event:
```

```
        # Handle GET request
```

```
        if (event['requestContext']['http']['method'] == "GET"):
```

```
            message = {
```

```
                'message': 'Execution started successfully!',
```

```
                'httpMethod': event['requestContext']['http']['method'],
```

```
                'context': context
```

```
            }
```

```
            response = {
```

```
                "statusCode": 200,
```

```
                "headers": {'Content-Type': 'application/json'},
```

```
                "body": json.dumps(message)
```

```
            }
```

```
            return response
```

```
        # Handle POST request
```

```
        elif (event['requestContext']['http']['method'] == "POST"):
```

```
            # loads the incoming event into a dictionary
```

```
            body = json.loads(event['body'])
```

```
            # Parse the input event
```

```
            question = body['question']
```

```
            context_text = body['context']
```

```
            # Call your BERT Q&A inference function here
```

```
            answer = bert_qa_inference(question, context_text)
```

```

        # Return the answer
        return {
            'statusCode': 200,
            'headers': {'Content-Type': 'application/json'},
            'body': answer
        }
    else:
        return {
            'statusCode': 200,
            'headers': {'Content-Type': 'application/json'},
            'body': json.dumps({'Event': event})
        }

def bert_qa_inference(question, context_text):
    # Encode the input text
    encoded_text = [question, context_text]

    try:
        # Initialize the SageMaker runtime client
        sagemaker_runtime = boto3.client('sagemaker-runtime')
        # Call the SageMaker endpoint
        response = sagemaker_runtime.invoke_endpoint(
            EndpointName=ENDPOINT_NAME,
            ContentType='application/list-text',
            Accept='application/json;verbose',
            Body=json.dumps(encoded_text)
        )

        response_body = json.loads(response['Body'].read().decode('utf-8'))
        answer = response_body['answer']

        return answer

    except Exception as e:
        error_message = f"An error occurred during BERT Q&A inference in  

Lambda function: {str(e)}"
        raise Exception(error_message)

```

## ДОДАТОК Б

```
#!/usr/bin/env python
# coding: utf-8
```

```
## Importing, fine-tuning and evaluating the XLM-roBERTa model on
Question Answering task
```

```
### 0. Importing the libraries
# ***
```

```
# In[2]:
```

```
get_ipython().system('pip install datasets')
get_ipython().system('pip install transformers')
get_ipython().system('pip install sentencepiece')
```

```
# In[3]:
```

```
get_ipython().run_line_magic('load_ext', 'autoreload')
get_ipython().run_line_magic('autoreload', '2')
```

```
# In[4]:
```

```
import os
import ast
import json
import time
# import pickle
import numpy as np
import pandas as pd
from tqdm.auto import tqdm
```

```
import multiprocessing
```

```
import boto3, json, sagemaker
from sagemaker.session import Session
```

```
from sagemaker.model import Model
from sagemaker.predictor import Predictor
from sagemaker.utils import name_from_base
from sagemaker.huggingface import HuggingFace
from sagemaker import image_uris, model_uris, script_uris
```

```
from datasets import load_dataset
from transformers import AutoTokenizer

from IPython.display import display, HTML

display(HTML("<style>.container { width:100% !important; }</style>"))

# In[ ]:

os.environ["TOKENIZERS_PARALLELISM"] = "false"

# In[15]:

sagemaker_session = sagemaker.Session()

# Specify the S3 bucket name and file path
sagemaker_session_bucket = 'question-answering-ukr-dataset'

if sagemaker_session_bucket is None and sagemaker_session is not None:
    # set to default bucket if a bucket name is not given
    sagemaker_session_bucket = sagemaker_session.default_bucket()

try:
    role = sagemaker.get_execution_role()
except ValueError:
    iam = boto3.client('iam')
    role = iam.get_role(RoleName='sagemaker_execution_role')['Role']['Arn']

aws_region = boto3.Session().region_name

sagemaker_session =
sagemaker.Session(default_bucket=sagemaker_session_bucket)

s3 = boto3.resource('s3')

# In[ ]:

# tokenizer used in preprocessing
tokenizer_name = 'deepset/xlm-roberta-base-squad2-distilled'

# ### Auxiliary functions
```

```
# In[24]:
```

```
def create_df_from_json(df: pd.DataFrame, type: str = None) -> pd.DataFrame:
    contexts = []
    questions = []
    is_impossible = []
    answers_text = []
    answers_start = []

    for i in range(df.shape[0]):
        topic = df.iloc[i, 1]['paragraphs']
        for sub_para in topic:
            for q_a in sub_para['qas']:
                if q_a['answers']:
                    for answer in q_a['answers']:
                        is_impossible.append(q_a['is_impossible'])
                        contexts.append(sub_para['context'])
                        questions.append(q_a['question'])
                        answers_text.append(answer['text'])
                        answers_start.append(answer['answer_start'])
                else:
                    is_impossible.append(q_a['is_impossible'])
                    contexts.append(sub_para['context'])
                    questions.append(q_a['question'])
                    answers_text.append([])
                    answers_start.append([])
            data = {
                "context": contexts,
                "question": questions,
                "is_impossible": is_impossible,
                "answer_start": answers_start,
                "answer_text": answers_text
            }
            output_df = pd.DataFrame(data)
    return output_df
```

```
# In[30]:
```

```
def process_row(row):

    idx, data = row
    real_answer = data['answer_text']
    start_idx = data['answer_start']
```

```

modified_data = data.copy() # Create a new dictionary

if isinstance(start_idx, list):
    return idx, modified_data

end_idx = start_idx + len(real_answer)

if data['context'][start_idx:end_idx] == real_answer:
    modified_data['answer_end'] = end_idx
elif data['context'][start_idx - 1:end_idx - 1] == real_answer:
    modified_data['answer_start'] = start_idx - 1
    modified_data['answer_end'] = end_idx - 1
elif data['context'][start_idx - 2:end_idx - 2] == real_answer:
    modified_data['answer_start'] = start_idx - 2
    modified_data['answer_end'] = end_idx - 2

return idx, modified_data

# In[ ]:

def create_end_idx(df):
    num_processes = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(processes=num_processes)

    rows = list(df.iterrows()) # Convert iterrows() to a list
    # rows = df.iterrows()
    # display(rows)
    result = pool.map(process_row, rows) # Pass the list to pool.map()
    # result = pool.map(process_row, df.iterrows())
    processed_rows = [(idx, data) for idx, data in result]

    pool.close()
    pool.join()

    output_df = pd.DataFrame([data for idx, data in processed_rows],
index=[str(idx) for idx, data in processed_rows], columns=df.columns)
    return output_df

# In[ ]:

def add_token_positions(encodings, answers):
    # initialize lists to contain the token indices of answer start/end

```

```

start_positions = []
end_positions = []

for idx in range(len(answers)):
    start_positions.append(encodings.char_to_token(idx,
answers[idx]['answer_start']))

    end_positions.append(encodings.char_to_token(idx,
answers[idx]['answer_end']))

    if start_positions[-1] is None:
        start_positions[-1] = tokenizer.model_max_length-1
    if end_positions[-1] is None:
        end_positions[-1] = tokenizer.model_max_length-1

    # update our encodings object with the new token-based start/end positions
    encodings.update({'start_positions': start_positions, 'end_positions':
end_positions})

# ## 1. Downloading the Squad2.0. dataset from the S3 AWS storage
# ---

# In[21]:

content_object = s3.Object('question-answering-ukr-dataset',
train_dataset+'/train-v2.0.json')
file_content = content_object.get()['Body'].read().decode('utf-8')
train_json_content = json.loads(file_content)

content_object = s3.Object('question-answering-ukr-dataset', test_dataset+'/dev-
v2.0.json')
file_content = content_object.get()['Body'].read().decode('utf-8')
val_json_content = json.loads(file_content)

# In[22]:

train = pd.DataFrame.from_dict(train_json_content)
test = pd.DataFrame.from_dict(val_json_content)

# In[23]:

train.shape, test.shape

```

```
# In[26]:
```

```
train_from_s3 = False
```

```
if train_from_s3:
```

```
    train_df_raw = create_df_from_json(train)
```

```
else:
```

```
    train_df_raw = pd.read_csv('train_df.csv', dtype={
```

```
        'context': 'string', 'question': 'string', 'is_impossible': 'bool', 'answer_text':  
'string'})
```

```
    train_df_raw.drop(columns={'Unnamed: 0'}, inplace=True)
```

```
# In[27]:
```

```
train_df = train_df_raw.copy()
```

```
# In[48]:
```

```
test_df_raw = create_df_from_json(test)
```

```
# In[50]:
```

```
test_df = test_df_raw.copy()
```

```
# In[51]:
```

```
display(train_df.head(2))
```

```
display(test_df.head(2))
```

```
# In[52]:
```

```
test_human_curreted = pd.read_json('dev-human-v2.0.json')
```

```
# In[53]:
```

```
test_human_curreted_df = create_df_from_json(test_human_curreted)
```

```
# In[56]:
```

```
train_df['answer_start'] = train_df['answer_start'].apply(lambda x:
```

```
ast.literal_eval(x) if x == '[]' else x)
```

```
test_df['answer_start'] = test_df['answer_start'].apply(lambda x:
```

```
ast.literal_eval(x) if x == '[]' else x)
```

```
test_human_curreted_df['answer_start'] =
test_human_curreted_df['answer_start'].apply(lambda x: ast.literal_eval(x) if x
== '[' else x)
```

```
# In[58]:
```

```
test_df['answer_start'] = test_df['answer_start'].apply(lambda row: np.nan if
isininstance(row, list) else row)
train_df['answer_start'] = train_df['answer_start'].apply(lambda row: np.nan if
isininstance(row, list) else row)
test_human_curreted_df['answer_start'] =
test_human_curreted_df['answer_start'].apply(lambda row: np.nan if
isininstance(row, list) else row)
```

```
# In[59]:
```

```
train_df = train_df.dropna(subset=['answer_start']).reset_index(drop=True)
print(train_df.is_impossible.value_counts())
```

```
test_df = test_df.dropna(subset=['answer_start']).reset_index(drop=True)
print(test_df.is_impossible.value_counts())
```

```
test_human_curreted_df =
test_human_curreted_df.dropna(subset=['answer_start']).reset_index(drop=True
)
print(test_human_curreted_df.is_impossible.value_counts())
```

```
# In[ ]:
```

```
train_df['is_str'] = [True if isinstance(row, str) else False for row in
train_df['answer_text']]
test_df['is_str'] = [True if isinstance(row, str) else False for row in
test_df['answer_text']]
```

```
# In[29]:
```

```
test_df['answer_end'] = 0
train_df['answer_end'] = 0
```

```
# In[31]:
```

```
start_time = time.time()
train_df_with_end = create_end_idx(train_df)
```

```
test_df_with_end = create_end_idx(test_df)
end_time = time.time()
print(abs(start_time - end_time))
```

*# Creating encodings for the model*

*# In[38]:*

```
tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)

train_encodings = tokenizer(train_df_with_end['context'].values.tolist(),
                             train_df_with_end['question'].values.tolist(),
                             truncation=True,
                             padding=True)
val_encodings = tokenizer(test_df_with_end['context'].values.tolist(),
                           test_df_with_end['question'].values.tolist(),
                           truncation=True,
                           padding=True)
```

*# In[ ]:*

```
train_df_only_possible[['answer_start', 'answer_end']] =
train_df_only_possible[['answer_start', 'answer_end']].astype(int)
```

```
test_df_only_possible[['answer_start', 'answer_end']] =
test_df_only_possible[['answer_start', 'answer_end']].astype(int)
```

*# In[ ]:*

```
test_df_only_possible['answers'] = test_df_only_possible.apply(lambda row: {
    'answer_text': row['answer_text'],
    'answer_start': row['answer_start'],
    'answer_end': row['answer_end']
}, axis=1)
```

```
train_df_only_possible['answers'] = train_df_only_possible.apply(lambda row:
{
    'answer_text': row['answer_text'],
    'answer_start': row['answer_start'],
    'answer_end': row['answer_end']
}, axis=1)
```

*# In[ ]:*

*# apply function to our data*

```
add_token_positions(train_encodings, train_df_only_possible['answers'])
add_token_positions(val_encodings, test_df_only_possible['answers'])
```

```
### 2. Creating an Estimator and start a training job
```

```
# ---
```

```
# In[8]:
```

```
model_name = 'deepset/xlm-roberta-base-squad2-distilled'
```

```
# In[ ]:
```

```
import datetime
ct = datetime.datetime.now()
current_time = str(ct.now()).replace(":", "-").replace(" ", "-")[:19]
training_job_name=f'finetune-{model_name}-{current_time}'
print( training_job_name )
```

```
# In[11]:
```

```
hyperparameters={'epochs': 3,
                  'train_batch_size': 32,
                  'model_name': model_name,
                  'tokenizer_name': tokenizer_name,
                  'output_dir': '/opt/ml/checkpoints',
                  }
```

```
# In[12]:
```

```
metric_definitions=[
    {'Name': 'loss', 'Regex': '"loss': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'learning_rate', 'Regex': '"learning_rate': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_loss', 'Regex': '"eval_loss': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_accuracy', 'Regex': '"eval_accuracy': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_f1', 'Regex': '"eval_f1': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_precision', 'Regex': '"eval_precision': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_recall', 'Regex': '"eval_recall': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_runtime', 'Regex': '"eval_runtime': ([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'eval_samples_per_second', 'Regex': '"eval_samples_per_second':
([0-9]+(\\.e\\-)[0-9]+),?"},
    {'Name': 'epoch', 'Regex': '"epoch': ([0-9]+(\\.e\\-)[0-9]+),?"}]
```

```
# In[19]:
```

```
huggingface_estimator = HuggingFace(
    entry_point='train.py',
    source_dir='./scripts',
    instance_type='ml.c5.xlarge',
    instance_count=1,
    checkpoint_s3_uri=f's3://{sagemaker_session_bucket}/models/checkpoints',
    use_spot_instances=True,
    role=role,
    transformers_version='4.26.0',
    pytorch_version='1.13.1',
    py_version='py39',
    hyperparameters = hyperparameters,
    metric_definitions=metric_definitions,
    max_run=36000, # expected max run in seconds
)
```

```
# In[ ]:
```

```
# starting the train job with our uploaded datasets as input
huggingface_estimator.fit({'train': 's3://question-answering-ukr-
dataset/datasets/squad-2.0+SDSJ-uk/encoded/train/train_encodings.pkl',
                          'test': 's3://question-answering-ukr-dataset/datasets/squad-
2.0+SDSJ-uk/encoded/train/test_encodings.pkl'},
                          wait=False, job_name=training_job_name)
```

```
# In[ ]:
```

```
sess.wait_for_job(training_job_name)
```

```
# ## 3. Training metrics
```

```
# ---
```

```
# In[ ]:
```

```
from sagemaker import TrainingJobAnalytics
```

```
# Captured metrics can be accessed as a Pandas dataframe
```

```
df = TrainingJobAnalytics(training_job_name=training_job_name).dataframe()
df.head(10)
```

```
# In[ ]:
```

```
evals = df[df.metric_name.isin(['eval_accuracy','eval_precision', 'eval_f1'])]
losses = df[df.metric_name.isin(['loss', 'eval_loss'])]
```

```
sns.lineplot(
    x='timestamp',
    y='value',
    data=evals,
    style='metric_name',
    markers=True,
    hue='metric_name'
)
```

```
ax2 = plt.twinx()
sns.lineplot(
    x='timestamp',
    y='value',
    data=losses,
    hue='metric_name',
    ax=ax2)
```

```
# ## 4. Deployment
```

```
# ---
```

```
# In[ ]:
```

```
predictor = huggingface_estimator.deploy(initial_instance_count=1,
instance_type="ml.m5.xlarge", endpoint_name=training_job_name)
```

## ДОДАТОК В

Допоміжні файли:

squad\_evaluation.py

"""Official evaluation script for SQuAD version 2.0.

In addition to basic functionality, we also compute additional statistics and plot precision-recall curves if an additional na\_prob.json file is provided. This file is expected to map question ID's to the model's predicted probability that a question is unanswerable.

"""

```
import argparse
import collections
import json
import numpy as np
import os
import re
import string
import sys
```

```
OPTS = None
```

```
def parse_args():
    parser = argparse.ArgumentParser(
        "Official evaluation script for SQuAD version 2.0."
    )
    parser.add_argument("data_file", metavar="data.json", help="Input data
JSON file.")
    parser.add_argument("pred_file", metavar="pred.json", help="Model
predictions.")
    parser.add_argument(
        "--out-file",
        "-o",
        metavar="eval.json",
        help="Write accuracy metrics to file (default is stdout).",
    )
    parser.add_argument(
        "--na-prob-file",
        "-n",
        metavar="na_prob.json",
        help="Model estimates of probability of no answer.",
    )
)
```

```

parser.add_argument(
    "--na-prob-thresh",
    "-t",
    type=float,
    default=1.0,
    help='Predict "" if no-answer probability exceeds this (default = 1.0).',
)
parser.add_argument(
    "--out-image-dir",
    "-p",
    metavar="out_images",
    default=None,
    help="Save precision-recall curves to directory.",
)
parser.add_argument("--verbose", "-v", action="store_true")
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(1)
return parser.parse_args()

```

```

def make_qid_to_has_ans(dataset):
    qid_to_has_ans = { }
    for article in dataset:
        for p in article["paragraphs"]:
            for qa in p["qas"]:
                qid_to_has_ans[qa["id"]] = bool(qa["answers"])
    return qid_to_has_ans

```

```

def normalize_answer(s):
    """Lower text and remove punctuation, articles and extra whitespace."""

```

```

def remove_articles(text):
    regex = re.compile(r"\b(a|an|the)\b", re.UNICODE)
    return re.sub(regex, " ", text)

```

```

def white_space_fix(text):
    return " ".join(text.split())

```

```

def remove_punc(text):
    exclude = set(string.punctuation)
    return "".join(ch for ch in text if ch not in exclude)

```

```

def lower(text):
    return text.lower()

return white_space_fix(remove_articles(remove_punc(lower(s))))

def get_tokens(s):
    if not s:
        return []
    return normalize_answer(s).split()

def compute_exact(a_gold, a_pred):
    return int(normalize_answer(a_gold) == normalize_answer(a_pred))

def compute_f1(a_gold, a_pred):
    gold_toks = get_tokens(a_gold)
    pred_toks = get_tokens(a_pred)
    common = collections.Counter(gold_toks) & collections.Counter(pred_toks)
    num_same = sum(common.values())
    if len(gold_toks) == 0 or len(pred_toks) == 0:
        # If either is no-answer, then F1 is 1 if they agree, 0 otherwise
        return int(gold_toks == pred_toks)
    if num_same == 0:
        return 0
    precision = 1.0 * num_same / len(pred_toks)
    recall = 1.0 * num_same / len(gold_toks)
    f1 = (2 * precision * recall) / (precision + recall)
    return f1

def get_raw_scores(dataset, preds):
    exact_scores = {}
    f1_scores = {}
    for article in dataset:
        for p in article["paragraphs"]:
            for qa in p["qas"]:
                qid = qa["id"]
                gold_answers = [
                    a["text"] for a in qa["answers"] if normalize_answer(a["text"])
                ]

```

```

    if not gold_answers:
        # For unanswerable questions, only correct answer is empty string
        gold_answers = [""]
    if qid not in preds:
        print("Missing prediction for %s" % qid)
        continue
    a_pred = preds[qid]
    # Take max over all gold answers
    exact_scores[qid] = max(compute_exact(a, a_pred) for a in
gold_answers)
    f1_scores[qid] = max(compute_f1(a, a_pred) for a in gold_answers)
return exact_scores, f1_scores

```

```

def apply_no_ans_threshold(scores, na_probs, qid_to_has_ans,
na_prob_thresh):
    new_scores = {}
    for qid, s in scores.items():
        pred_na = na_probs[qid] > na_prob_thresh
        if pred_na:
            new_scores[qid] = float(not qid_to_has_ans[qid])
        else:
            new_scores[qid] = s
    return new_scores

```

```

def make_eval_dict(exact_scores, f1_scores, qid_list=None):
    if not qid_list:
        total = len(exact_scores)
        return collections.OrderedDict(
            [
                ("exact", 100.0 * sum(exact_scores.values()) / total),
                ("f1", 100.0 * sum(f1_scores.values()) / total),
                ("total", total),
            ]
        )
    else:
        total = len(qid_list)
        return collections.OrderedDict(
            [
                ("exact", 100.0 * sum(exact_scores[k] for k in qid_list) / total),
                ("f1", 100.0 * sum(f1_scores[k] for k in qid_list) / total),
                ("total", total),
            ]
        )

```

```
]
)
```

```
def merge_eval(main_eval, new_eval, prefix):
    for k in new_eval:
        main_eval["%s_%s" % (prefix, k)] = new_eval[k]
```

```
def plot_pr_curve(precisions, recalls, out_image, title):
    plt.step(recalls, precisions, color="b", alpha=0.2, where="post")
    plt.fill_between(recalls, precisions, step="post", alpha=0.2, color="b")
    plt.xlabel("Recall")
    plt.ylabel("Precision")
    plt.xlim([0.0, 1.05])
    plt.ylim([0.0, 1.05])
    plt.title(title)
    plt.savefig(out_image)
    plt.clf()
```

```
def make_precision_recall_eval(
    scores, na_probs, num_true_pos, qid_to_has_ans, out_image=None,
    title=None
):
    qid_list = sorted(na_probs, key=lambda k: na_probs[k])
    true_pos = 0.0
    cur_p = 1.0
    cur_r = 0.0
    precisions = [1.0]
    recalls = [0.0]
    avg_prec = 0.0
    for i, qid in enumerate(qid_list):
        if qid_to_has_ans[qid]:
            true_pos += scores[qid]
            cur_p = true_pos / float(i + 1)
            cur_r = true_pos / float(num_true_pos)
            if i == len(qid_list) - 1 or na_probs[qid] != na_probs[qid_list[i + 1]]:
                # i.e., if we can put a threshold after this point
                avg_prec += cur_p * (cur_r - recalls[-1])
            precisions.append(cur_p)
            recalls.append(cur_r)
    if out_image:
```

```
    plot_pr_curve(precisions, recalls, out_image, title)
return {"ap": 100.0 * avg_prec }
```

```
def run_precision_recall_analysis(
    main_eval, exact_raw, f1_raw, na_probs, qid_to_has_ans, out_image_dir
):
    if out_image_dir and not os.path.exists(out_image_dir):
        os.makedirs(out_image_dir)
    num_true_pos = sum(1 for v in qid_to_has_ans.values() if v)
    if num_true_pos == 0:
        return
    pr_exact = make_precision_recall_eval(
        exact_raw,
        na_probs,
        num_true_pos,
        qid_to_has_ans,
        out_image=os.path.join(out_image_dir, "pr_exact.png"),
        title="Precision-Recall curve for Exact Match score",
    )
    pr_f1 = make_precision_recall_eval(
        f1_raw,
        na_probs,
        num_true_pos,
        qid_to_has_ans,
        out_image=os.path.join(out_image_dir, "pr_f1.png"),
        title="Precision-Recall curve for F1 score",
    )
    oracle_scores = {k: float(v) for k, v in qid_to_has_ans.items()}
    pr_oracle = make_precision_recall_eval(
        oracle_scores,
        na_probs,
        num_true_pos,
        qid_to_has_ans,
        out_image=os.path.join(out_image_dir, "pr_oracle.png"),
        title="Oracle Precision-Recall curve (binary task of HasAns vs. NoAns)",
    )
    merge_eval(main_eval, pr_exact, "pr_exact")
    merge_eval(main_eval, pr_f1, "pr_f1")
    merge_eval(main_eval, pr_oracle, "pr_oracle")
```

```
def histogram_na_prob(na_probs, qid_list, image_dir, name):
```

```

if not qid_list:
    return
x = [na_probs[k] for k in qid_list]
weights = np.ones_like(x) / float(len(x))
plt.hist(x, weights=weights, bins=20, range=(0.0, 1.0))
plt.xlabel("Model probability of no-answer")
plt.ylabel("Proportion of dataset")
plt.title("Histogram of no-answer probability: %s" % name)
plt.savefig(os.path.join(image_dir, "na_prob_hist_%s.png" % name))
plt.clf()

```

```

def find_best_thresh(preds, scores, na_probs, qid_to_has_ans):
    num_no_ans = sum(1 for k in qid_to_has_ans if not qid_to_has_ans[k])
    cur_score = num_no_ans
    best_score = cur_score
    best_thresh = 0.0
    qid_list = sorted(na_probs, key=lambda k: na_probs[k])
    for i, qid in enumerate(qid_list):
        if qid not in scores:
            continue
        if qid_to_has_ans[qid]:
            diff = scores[qid]
        else:
            if preds[qid]:
                diff = -1
            else:
                diff = 0
        cur_score += diff
        if cur_score > best_score:
            best_score = cur_score
            best_thresh = na_probs[qid]
    return 100.0 * best_score / len(scores), best_thresh

```

```

def find_all_best_thresh(main_eval, preds, exact_raw, f1_raw, na_probs,
qid_to_has_ans):
    best_exact, exact_thresh = find_best_thresh(
        preds, exact_raw, na_probs, qid_to_has_ans
    )
    best_f1, f1_thresh = find_best_thresh(preds, f1_raw, na_probs,
qid_to_has_ans)
    main_eval["best_exact"] = best_exact

```

```
main_eval["best_exact_thresh"] = exact_thresh
main_eval["best_f1"] = best_f1
main_eval["best_f1_thresh"] = f1_thresh
```

```
def main():
    with open(OPTS.data_file) as f:
        dataset_json = json.load(f)
        dataset = dataset_json["data"]
    with open(OPTS.pred_file) as f:
        preds = json.load(f)
    if OPTS.na_prob_file:
        with open(OPTS.na_prob_file) as f:
            na_probs = json.load(f)
    else:
        na_probs = {k: 0.0 for k in preds}
    qid_to_has_ans = make_qid_to_has_ans(dataset) # maps qid to True/False
    has_ans_qids = [k for k, v in qid_to_has_ans.items() if v]
    no_ans_qids = [k for k, v in qid_to_has_ans.items() if not v]
    exact_raw, f1_raw = get_raw_scores(dataset, preds)
    exact_thresh = apply_no_ans_threshold(
        exact_raw, na_probs, qid_to_has_ans, OPTS.na_prob_thresh
    )
    f1_thresh = apply_no_ans_threshold(
        f1_raw, na_probs, qid_to_has_ans, OPTS.na_prob_thresh
    )
    out_eval = make_eval_dict(exact_thresh, f1_thresh)
    if has_ans_qids:
        has_ans_eval = make_eval_dict(exact_thresh, f1_thresh,
qid_list=has_ans_qids)
        merge_eval(out_eval, has_ans_eval, "HasAns")
    if no_ans_qids:
        no_ans_eval = make_eval_dict(exact_thresh, f1_thresh,
qid_list=no_ans_qids)
        merge_eval(out_eval, no_ans_eval, "NoAns")
    if OPTS.na_prob_file:
        find_all_best_thresh(
            out_eval, preds, exact_raw, f1_raw, na_probs, qid_to_has_ans
        )
    if OPTS.na_prob_file and OPTS.out_image_dir:
        run_precision_recall_analysis(
            out_eval, exact_raw, f1_raw, na_probs, qid_to_has_ans,
OPTS.out_image_dir
```

```

    )
    histogram_na_prob(na_probs, has_ans_qids, OPTS.out_image_dir,
"hasAns")
    histogram_na_prob(na_probs, no_ans_qids, OPTS.out_image_dir,
"noAns")
    if OPTS.out_file:
        with open(OPTS.out_file, "w") as f:
            json.dump(out_eval, f)
    else:
        print(json.dumps(out_eval, indent=2))

```

```

if __name__ == "__main__":
    OPTS = parse_args()
    if OPTS.out_image_dir:
        import matplotlib

        matplotlib.use("Agg")
        import matplotlib.pyplot as plt
    main()

```

train.py

```

"""

```

Training script for Hugging Face SageMaker Estimator

```

"""

```

```

import logging
import sys
import argparse
import os
from transformers import AutoModelForSequenceClassification,
AutoTokenizer
from transformers import Trainer, TrainingArguments
from datasets import load_from_disk
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

```

```

if __name__ == "__main__":

```

```

    parser = argparse.ArgumentParser()

```

*# hyperparameters sent by the client are passed as command-line arguments to the script.*

```

    parser.add_argument("--epochs", type=int, default=3)
    parser.add_argument("--train_batch_size", type=int, default=32)

```

```

parser.add_argument("--eval_batch_size", type=int, default=64)
parser.add_argument("--warmup_steps", type=int, default=500)
parser.add_argument("--model_name", type=str)
parser.add_argument("--tokenizer_name", type=str)
parser.add_argument("--learning_rate", type=str, default=5e-5)

# Data, model, and output directories
parser.add_argument("--output-data-dir", type=str,
default=os.environ["SM_OUTPUT_DATA_DIR"])
parser.add_argument("--model-dir", type=str,
default=os.environ["SM_MODEL_DIR"])
parser.add_argument("--n_gpus", type=str,
default=os.environ["SM_NUM_GPUS"])
parser.add_argument("--training_dir", type=str,
default=os.environ["SM_CHANNEL_TRAIN"])
parser.add_argument("--test_dir", type=str,
default=os.environ["SM_CHANNEL_TEST"])

args, _ = parser.parse_known_args()

# Set up logging
logger = logging.getLogger(__name__)

logging.basicConfig(
    level=logging.getLevelName("INFO"),
    handlers=[logging.StreamHandler(sys.stdout)],
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)

# load datasets
train_dataset = load_from_disk(args.training_dir)
test_dataset = load_from_disk(args.test_dir)

logger.info("loaded train_dataset length is: %s", len(train_dataset))
logger.info("loaded test_dataset length is: %s", len(test_dataset))

def compute_metrics(pred):
    """Compute metrics function for binary classification"""
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f_1, _ = precision_recall_fscore_support(labels, preds,
average="binary")
    acc = accuracy_score(labels, preds)

```

```

    return {"accuracy": acc, "f1": f_1, "precision": precision, "recall": recall}

# download model and tokenizer from model hub
model =
AutoModelForSequenceClassification.from_pretrained(args.model_name)
tokenizer = AutoTokenizer.from_pretrained(args.tokenizer_name)

# define training args
training_args = TrainingArguments(
    output_dir=args.model_dir,
    num_train_epochs=args.epochs,
    per_device_train_batch_size=args.train_batch_size,
    per_device_eval_batch_size=args.eval_batch_size,
    warmup_steps=args.warmup_steps,
    evaluation_strategy="epoch",
    logging_dir=f"{args.output_data_dir}/logs",
    learning_rate=float(args.learning_rate),
)

# create Trainer instance
trainer = Trainer(
    model=model,
    args=training_args,
    compute_metrics=compute_metrics,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=tokenizer,
)

# train model
trainer.train()

# evaluate model
eval_result = trainer.evaluate(eval_dataset=test_dataset)

# writes eval result to file which can be accessed later in s3 ouput
with open(os.path.join(args.output_data_dir, "eval_results.txt"), "w") as
writer:
    print("***** Eval results *****")
    for key, value in sorted(eval_result.items()):
        writer.write(f"{key} = {value}\n")

# Saves the model to s3

```

```
trainer.save_model(args.model_dir)
```