	Creating a Sentiment Analysis Web App Using PyTorch and SageMaker Deep Learning Nanodegree Program Deployment Now that we have a basic understanding of how SageMaker works we will try to use it to construct a complete project from end to end. Our goal will be to have a simple web page which a user can use to enter a movie review. The web page will then send the review off to our deployed model which will predict the sentiment of the entered review. Instructions
	Some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this notebook. You will not need to modify the included code beyond what is requested. Sections that begin with 'TODO' in the header indicate that you need to complete or implement some portion within them. Instructions will be provided for each section and the specifics of the implementation are marked in the code block with a # TODO: comment. Please be sure to read the instructions carefully! In addition to implementing code, there will be questions for you to answer which relate to the task and your implementation. Each section where you will answer a question is preceded by a 'Question:' header. Carefully read each question and provide your answer below the 'Answer:' header by editing the Markdown cell. Note: Code and Markdown cells can be executed using the Shift+Enter keyboard shortcut. In addition, a cell can be edited by typically clicking it (double-click for Markdown cells) or by pressing Enter while it is highlighted.
	By typically clicking it (double-click for Markdown cells) or by pressing Enter while it is highlighted. General Outline Recall the general outline for SageMaker projects using a notebook instance. 1. Download or otherwise retrieve the data. 2. Process / Prepare the data. 3. Upload the processed data to S3. 4. Train a chosen model. 5. Test the trained model (typically using a batch transform job).
	 6. Deploy the trained model. 7. Use the deployed model. For this project, you will be following the steps in the general outline with some modifications. First, you will not be testing the model in its own step. You will still be testing the model, however, you will do it by deploying your model and then using the deployed model by sending the test data to it. One of the reasons for doing this is so that you can make sure that your deployed model is working correctly before moving forward. In addition, you will deploy and use your trained model a second time. In the second iteration you will customize the way that your trained model is deployed by including some of your own code. In addition, your newly deployed model will be used in the sentiment analysis web app.
In [2]:	<pre># Make sure that we use SageMaker 1.x !pip install sagemaker==1.72.0 INSTALLED ALREADY Usage: pip install [options] <requirement specifier=""> [package-index-options] pip install [options] -r <requirements file=""> [package-index-options] pip install [options] [-e] <vcs project="" url=""> pip install [options] [-e] <local path="" project=""> pip install [options] <archive path="" url=""> no such option:</archive></local></vcs></requirements></requirement></pre>
In [3]:	Step 1: Downloading the data As in the XGBoost in SageMaker notebook, we will be using the IMDb dataset Maas, Andrew L., et al. Learning Word Vectors for Sentiment Analysis. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, 2011.
	<pre>!wget -0/data/aclImdb_v1.tar.gz http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz !tar -zxf/data/aclImdb_v1.tar.gz -C/data mkdir: cannot create directory '/data': File exists2020-12-30 12:11:35 http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz Resolving ai.stanford.edu (ai.stanford.edu) 171.64.68.10 Connecting to ai.stanford.edu (ai.stanford.edu) 171.64.68.10 :80 connected. HTTP request sent, awaiting response 200 OK Length: 84125825 (80M) [application/x-gzip] Saving to: '/data/aclImdb_v1.tar.gz'/data/aclImdb_v1. 100%[===================================</pre>
In [4]:	Step 2: Preparing and Processing the data Also, as in the XGBoost notebook, we will be doing some initial data processing. The first few steps are the same as in the XGBoost example. To begin with, we will read in each of the reviews and combine them into a single input structure. Then, we will split the dataset into a training set and a testing set.
	<pre>import glob def read_imdb_data(data_dir='/data/aclImdb'): data = {} labels = {} for data_type in ['train', 'test']: data[data_type] = {} labels[data_type] = {} for sentiment in ['pos', 'neg']: data[data_type][sentiment] = [] labels[data_type][sentiment] = []</pre>
	<pre>path = os.path.join(data_dir, data_type, sentiment, '*.txt') files = glob.glob(path) for f in files: with open(f) as review: data[data_type][sentiment].append(review.read()) # Here we represent a positive review by '1' and a negative review by '0' labels[data_type][sentiment].append(1 if sentiment == 'pos' else 0) assert len(data[data_type][sentiment]) == len(labels[data_type][sentiment]), \ "{}/{} data size does not match labels size".format(data_type, sentiment)</pre>
In [5]:	<pre>data, labels = read_imdb_data() print("IMDB reviews: train = {} pos / {} neg, test = {} pos / {} neg".format(</pre>
In [6]:	<pre>def prepare_imdb_data(data, labels): """Prepare training and test sets from IMDb movie reviews.""" #Combine positive and negative reviews and labels data_train = data['train']['pos'] + data['train']['neg'] data_test = data['test']['pos'] + data['test']['neg'] labels_train = labels['train']['pos'] + labels['train']['neg'] labels_test = labels['test']['pos'] + labels['test']['neg'] #Shuffle reviews and corresponding labels within training and test sets</pre>
In [7]:	<pre>data_train, labels_train = shuffle(data_train, labels_train) data_test, labels_test = shuffle(data_test, labels_test) # Return a unified training data, test data, training labels, test labets return data_train, data_test, labels_train, labels_test train_X, test_X, train_y, test_y = prepare_imdb_data(data, labels) print("IMDb reviews (combined): train = {}, test = {}".format(len(train_X), len(test_X))) IMDb reviews (combined): train = 25000, test = 25000</pre> Now that we have our training and testing sets unified and prepared, we should do a quick check and see an example of the data our model "The labels_train = shuffle(data_train, labels_train) # Return a unified training data, test data, training labels, test labets return data_train, data_test, labels_train, labels_test # Return a unified training data, test data, training labels, test labets return data_train, data_test, labels_train, labels_test labels_test labets return data_train, data_test, labels_test labels_test labets # Return a unified training data, test data, training labels, test labets return data_train, data_test, labels_train, labels_test labels_test labets return data_train, data_test, labels_test labels_test labets return data_train, data_test, labels_test labels_test labets return data_train, data_test labels_test labels_test labets return data_test labets labels_test labets return data_test labets labels_test labets labets labets labels_test labets labels_test labets labels_test lab
In [8]:	will be trained on. This is generally a good idea as it allows you to see how each of the further processing steps affects the reviews and it also ensures that the data has been loaded correctly. print(train_X[100]) print(train_y[100]) Average (and surprisingly tame) Fulci giallo which means it's still quite bad by normal standards, bu t redeemed by its solid build-up and some nice touches such as a neat time twist on the issues of vis ions and clairvoyance. by />cbr />The genre's well-known weaknesses are in full gear: banal dialogue, wooden acting, illogical plot points. And the finale goes on much too long, while the denouement proves to be a rather lame or shall I say: limp affair. s is amusing, though. Yellow clues wherever you look. />cbr />cbr />sbr />s out of 10 limping killers
In [9]:	The first step in processing the reviews is to make sure that any html tags that appear should be removed. In addition we wish to tokenize our input, that way words such as entertained and entertaining are considered the same with regard to sentiment analysis. import nltk from nltk.corpus import stopwords from nltk.stem.porter import * import re from bs4 import BeautifulSoup
	<pre>def review_to_words(review): nltk.download("stopwords", quiet=True) stemmer = PorterStemmer() text = BeautifulSoup(review, "html.parser").get_text() # Remove HTML tags text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower()) # Convert to lower case words = text.split() # Split string into words words = [w for w in words if w not in stopwords.words("english")] # Remove stopwords words = [PorterStemmer().stem(w) for w in words] # stem return words</pre>
<pre>In [10]: Out[10]:</pre>	<pre>review_to_words(train_X[400]) #We can see its working proeprly ['unusu', 'straight', 'face', 'action', 'play',</pre>
	<pre>'cast', 'film', 'director', 'obvious', 'took', 'materi', 'serious', 'imperfect', 'expect', 'film', 'clearli', 'shot', 'tight',</pre>
	<pre>'budget', 'drama', 'involv', 'one', 'film', 'get', 'repeat', 'ad', 'nauseum', 'cinemax', '2', 'max', 'whatev',</pre>
	<pre>'call', 'end', 'watch', '40', 'minut', 'block', 'suppos', 'go', 'work', 'along', 'w', 'deathstalk', '2',</pre>
	<pre>'chop', 'mall', 'assault', 'remind', 'wynorski', 'much', 'talent', 'director', 'mani', 'fellow', 'low', 'budget', 'brethern',</pre>
	<pre>'brethern', 'real', 'abil', 'pace', 'genr', 'film', 'actual', 'interest', 'materi', 'e', 'bother', 'watch', 'shannon', 'tweed', 'flick'.</pre>
	<pre>'flick', '3', '4', 'titl', 'actor', 'littl', 'recent', 'mancuso', 'ford', 'even', 'gari', 'sandi', 'chrissak',</pre>
	<pre>'realli', 'put', 'best', 'role', 'year', 'grieco', 'right', 'look', 'although', 'act', 'bit', 'one', 'note',</pre>
	<pre>'clear', 'charact', 'suppos', 'self', 'destruct', 'throughout', 'film', 'grieco', 'quit', 'convey', 'check', 'imdb', 'see',</pre>
	<pre>'writer', 'also', 'wrote', 'soror', 'hous', 'massacr', '2', 'dinosaur', 'island', 'director', 'minor', 'classic', 'right',</pre>
	<pre>'obvious', 'silli', 'roger', 'cormon', 'like', 'cinema', 'one', 'like', 'better', 'jonathan', 'demm', 'jonathan',</pre>
	<pre>'kaplan', 'b', 'pictur', '70', 'give', 'exploit', 'element', 'offer', 'involv', 'drama', 'time', 'real', 'step',</pre>
	<pre>'forward', 'citizen', 'kane', 'comic', 'final', 'moment', 'bit', 'disrupt', 'well', 'written', 'charact', 'driven', 'averag',</pre>
	<pre>'straight', 'video', 'action', 'small', 'achiev', 'like', 'overlook', 'come', 'along', 'rare', 'enough', 'remind', 'tri',</pre>
	<pre>'sit', 'albert', 'pyun', 'monstros', 'call', 'heatseek', 'night', 'low', 'budget', 'stuff', 'easi', 'look', 'anoth',</pre>
	Question: Above we mentioned that <pre>review_to_words</pre> method removes html formatting and allows us to tokenize the words found in a review, for example, converting entertained and entertaining into entertain so that they are treated as though they are the same word. What else, if anything, does this method do to the input? Answer: This does a few other stuffs to the input such as removing HTML tags and also ensuring that all letters are converted to lowercase. The method below applies the <pre>review_to_words</pre> method to each of the reviews in the training and testing datasets. In addition it caches the results. This is because performing this processing step can take a long time. This way if you are unable to complete the notebook in the
	<pre>def preprocess_data(data_train, data_test, labels_train, labels_test,</pre>
In [12]:	# Preprocess data train_X, test_X, train_y, test_y = preprocess_data(train_X, test_X, train_y, test_y) Read preprocessed data from cache file: preprocessed_data.pkl Transform the data In the XGBoost notebook we transformed the data from its word representation to a bag-of-words feature representation. For the model we are going to construct in this notebook we will construct a feature representation which is very similar. To start, we will represent each word as an integer. Of course, some of the words that appear in the reviews occur very infrequently and so likely don't contain much information for the purposes of sentiment analysis. The way we will deal with this problem is that we will fix the size of our working vocabulary and we will only include the words that appear most frequently. We will then combine all of the infrequent words into a single category and, in our case, we will label it as 1. Since we will be using a recurrent neural network, it will be convenient if the length of each review is the same. To do this, we will fix a size
	Since we will be using a recurrent neural network, it will be convenient if the length of each review is the same. To do this, we will fix a size for our reviews and then pad short reviews with the category 'no word' (which we will label 0) and truncate long reviews. (TODO) Create a word dictionary To begin with, we need to construct a way to map words that appear in the reviews to integers. Here we fix the size of our vocabulary (including the 'no word' and 'infrequent' categories) to be 5000 but you may wish to change this to see how it affects the model. TODO: Complete the implementation for the build_dict() method below. Note that even though the vocab_size is set to 5000, we only want to construct a mapping for the most frequently appearing 4998 words. This is because we want to reserve the special labels 0 for 'no word' and 1 for 'infrequent word'.
In [13]:	<pre>import numpy as np from collections import defaultdict def build_dict(data, vocab_size = 5000): """Construct and return a dictionary mapping each of the most frequently appearing words to a unique integer.""" # TODO: Determine how often each word appears in `data`. Note that `data` is a list of sentences and that a # sentence is a list of words.</pre>
	<pre>from collections import defaultdict word_count = defaultdict(int) # This allows to count up the amount of time a word occurs withut w orrying about KeyError where it provides a default value if the key doesnt exist for row in data: for word in row: word_count[word] += 1 word_count= dict(word_count) #Convert to dictionary</pre>
In [14]:	<pre>sorted_words = [word for word, count in sorted(word_count.items(), key= lambda pair:pair[1], revers e=True)] word_dict = {} # This is what we are building, a dictionary that translates words into integers for idx, word in enumerate(sorted_words[:vocab_size - 2]): # The -2 is so that we save room for the 'no word' word_dict[word] = idx + 2 # 'infrequent' labels return word_dict word_dict = build_dict(train_X)</pre>
<pre>In [15]: Out[15]:</pre>	Question: What are the five most frequently appearing (tokenized) words in the training set? Does it makes sense that these words appear frequently in the training set? Answer: Extracting the first five frequent words gives us 'movi', 'film', 'one', 'like', 'time'. Yes sicne we are clearly working with IMDB dataset which consists of movie reviews it makes sense in that regard. # TODO: Use this space to determine the five most frequently appearing words in the training set. list(word_dict)[0:5] #Index the first 5 frequent words ['movi', 'film', 'one', 'like', 'time']
In [16]:	Later on when we construct an endpoint which processes a submitted review we will need to make use of the word_dict which we have created. As such, we will save it to a file now for future use. data_dir = '/data/pytorch' # The folder we will use for storing data if not os.path.exists(data_dir): # Make sure that the folder exists os.makedirs(data_dir) with open(os.path.join(data_dir, 'word_dict.pkl'), "wb") as f: pickle.dump(word_dict, f)
In [18]:	Transform the reviews Now that we have our word dictionary which allows us to transform the words appearing in the reviews into integers, it is time to make use of it and convert our reviews to their integer sequence representation, making sure to pad or truncate to a fixed length, which in our case is 500. def convert_and_pad(word_dict, sentence, pad=500): NOWORD = 0 # We will use 0 to represent the 'no word' category INFREQ = 1 # and we use 1 to represent the infrequent words, i.e., words not appearing in word_dict
	<pre>working_sentence = [NOWORD] * pad for word_index, word in enumerate(sentence[:pad]): if word in word_dict: working_sentence[word_index] = word_dict[word] else: working_sentence[word_index] = INFREQ return working_sentence, min(len(sentence), pad) def convert_and_pad_data(word_dict, data, pad=500): result = [] lengths = []</pre>
In [19]:	<pre>for sentence in data: converted, leng = convert_and_pad(word_dict, sentence, pad) result.append(converted) lengths.append(leng) return np.array(result), np.array(lengths) train_X, train_X_len = convert_and_pad_data(word_dict, train_X) test_X, test_X_len = convert_and_pad_data(word_dict, test_X)</pre> As a quick check to make sure that things are working as intended, check to see what one of the reviews in the training set looks like after
In [20]:	having been processeed. Does this look reasonable? What is the length of a review in the training set? # Use this cell to examine one of the processed reviews to make sure everything is working as intended. print(train_X[1]) #First sample case print(len(train_X[1])) #Confirming if it is indeed 500 [671 573 394 329 48 566 2 25 11 249 202 77 219 516 10 584 25 2 651 223 208 209 252 48 1338 11 2 2315 249 1826 11 166 290 841 536 25 110 1820 2 26 1012 70 6 123 479 269 60 70 6 11 2 43 16 725 2 20 131 30 1345 60 21 5 30 421 60 326 1072 2 269 1293 1547 3486 12 2 146 1163 547 566 81 1851 2554 43 30 141
	Question: In the cells above we use the preprocess_data and convert_and_pad_data methods to process both the training and testing set. Why or why not might this be a problem? Answer: This shouldn't be a problem to the best of my knowlege as we want to ensure everything is consistent, in other words everything which is being applied to the training data is also done to the testing set. So as just went through in summary the first method
	which is being applied to the training data is also done to the testing set. So as just went through in summary the first method prepocess_data which just applies the review_to_words method to all our reviews in both training and testing set. Lastly, our convert_and_pad ensures to convert our reviews to their integer sequence representation and also making sure to pad or truncate to a fixed length in our case 500. Step 3: Upload the data to S3 As in the XGBoost notebook, we will need to upload the training dataset to S3 in order for our training code to access it. For now we will save it locally and we will upload to S3 later on. Save the processed training dataset locally
In [21]:	It is important to note the format of the data that we are saving as we will need to know it when we write the training code. In our case, each row of the dataset has the form <code>label</code> , <code>length</code> , <code>review[500]</code> where <code>review[500]</code> is a sequence of <code>500</code> integers representing the words in the review.
In [22]:	Next, we need to upload the training data to the SageMaker default S3 bucket so that we can provide access to it while training our model.
In [23]:	NOTE: The cell above uploads the entire contents of our data directory. This includes the word_dict.pkl file. This is fortunate as we will need this later on when we create an endpoint that accepts an arbitrary review. For now, we will just take note of the fact that it resides in the data directory (and so also in the S3 training bucket) and that we will need to make sure it gets saved in the model directory. Step 4: Build and Train the PyTorch Model In the XGBoost notebook we discussed what a model is in the SageMaker framework. In particular, a model comprises three objects
	 Model Artifacts, Training Code, and Inference Code, each of which interact with one another. In the XGBoost example we used training and inference code that was provided by Amazon. Here we will still be using containers provided by Amazon with the added benefit of being able to include our own custom code. We will start by implementing our own neural network in PyTorch along with a training script. For the purposes of this project we have provided the necessary model object in the model.py file, inside of the train folder. You can see the provided implementation by running the cell below.
In [24]:	<pre>!pygmentize train/model.py import torch.nn as nn class LSTMClassifier(nn.Module): """ This is the simple RNN model we will be using to perform Sentiment Analysis. """ definit(self, embedding_dim, hidden_dim, vocab_size):</pre>
	<pre>super(LSTMClassifier, self)init() self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0) self.lstm = nn.LSTM(embedding_dim, hidden_dim) self.dense = nn.Linear(in_features=hidden_dim, out_features=1) self.sig = nn.Sigmoid() self.word_dict = None def forward(self, x): """ Perform a forward pass of our model on some input.</pre>
	<pre>x = x.t() lengths = x[0,:] reviews = x[1:,:] embeds = self.embedding(reviews) lstm_out, _ = self.lstm(embeds) out = self.dense(lstm_out) out = out[lengths - 1, range(len(lengths))] return self.sig(out.squeeze())</pre> The important takeaway from the implementation provided is that there are three parameters that we may wish to tweak to improve the performance of our model. These are the embedding dimension, the hidden dimension and the size of the vocabulary. We will likely want to
In [25]:	make these parameters configurable in the training script so that if we wish to modify them we do not need to modify the script itself. We will see how to do this later on. To start we will write some of the training code in the notebook so that we can more easily diagnose any issues that arise. First we will load a small portion of the training data set to use as a sample. It would be very time consuming to try and train the model completely in the notebook as we do not have access to a gpu and the compute instance that we are using is not particularly powerful. However, we can work on a small bit of the data to get a feel for how our training script is behaving.
	<pre>train_sample = pd.read_csv(os.path.join(data_dir, 'train.csv'), header=None, names=None, nrows=250) # Turn the input pandas dataframe into tensors train_sample_y = torch.from_numpy(train_sample[[0]].values).float().squeeze() train_sample_X = torch.from_numpy(train_sample.drop([0], axis=1).values).long() # Build the dataset train_sample_ds = torch.utils.data.TensorDataset(train_sample_X, train_sample_y) # Build the dataloader train_sample_dl = torch.utils.data.DataLoader(train_sample_ds, batch_size=50)</pre>
In [26]:	(TODO) Writing the training method Next we need to write the training code itself. This should be very similar to training methods that you have written before to train PyTorch models. We will leave any difficult aspects such as model saving / loading and parameter loading until a little later. def train(model, train_loader, epochs, optimizer, loss_fn, device): for epoch in range(1, epochs + 1): model.train() total_loss = 0 for batch in train_loader: batch_X, batch_y = batch batch_X = batch_X.to(device)
	<pre>batch_X = batch_X.to(device) batch_y = batch_y.to(device) # TODO: Complete this train method to train the model provided. optimizer.zero_grad() output = model.forward(batch_X) #Forward loss = loss_fn(output, batch_y) #Applying loss fxn loss.backward() #Backword optimizer.step() total_loss += loss.data.item() print("Epoch: {}, BCELoss: {}".format(epoch, total_loss / len(train_loader)))</pre>
In [27]:	Supposing we have the training method above, we will test that it is working by writing a bit of code in the notebook that executes our training method on the small sample training set that we loaded earlier. The reason for doing this in the notebook is so that we have an opportunity to fix any errors that arise early when they are easier to diagnose. import torch.optim as optim from train.model import LSTMClassifier device = torch.device("cuda" if torch.cuda.is_available() else "cpu") model = LSTMClassifier(32, 100, 5000).to(device) optimizer = optim.Adam(model.parameters()) loss_fn = torch.nn.BCELoss()
	train(model, train_sample_dl, 5, optimizer, loss_fn, device) Epoch: 1, BCELoss: 0.6964563131332397 Epoch: 2, BCELoss: 0.6871204495429992 Epoch: 3, BCELoss: 0.6801021814346313 Epoch: 4, BCELoss: 0.6729938507080078 Epoch: 5, BCELoss: 0.6650840997695923 In order to construct a PyTorch model using SageMaker we must provide SageMaker with a training script. We may optionally include a directory which will be copied to the container and from which our training code will be run. When the training container is executed it will check the uploaded directory (if there is one) for a requirements.txt file and install any required Python libraries, after which the
	(TODO) Training the model When a PyTorch model is constructed in SageMaker, an entry point must be specified. This is the Python file which will be executed when the model is trained. Inside of the train directory is a file called train.py which has been provided and which contains most of the necessary code to train our model. The only thing that is missing is the implementation of the train() method which you wrote earlier in this notebook. TODO: Copy the train() method written above and paste it into the train/train.py file where required.
	The way that SageMaker passes hyperparameters to the training script is by way of arguments. These arguments can then be parsed and used in the training script. To see how this is done take a look at the provided train.py file.

In [28]: from sagemaker.pytorch import PyTorch estimator = PyTorch(entry point="train.py", source dir="train", role=role, framework version='0.4.0', train instance count=1, train instance type='ml.p2.xlarge', hyperparameters={ 'epochs': 10, 'hidden dim': 200, In [29]: estimator.fit({'training': input data}) 'create image uri' will be deprecated in favor of 'ImageURIProvider' class in SageMaker Python SDK v 2. 's3 input' class will be renamed to 'TrainingInput' in SageMaker Python SDK v2. 'create image uri' will be deprecated in favor of 'ImageURIProvider' class in SageMaker Python SDK v 2020-12-30 12:14:19 Starting - Starting the training job... 2020-12-30 12:14:21 Starting - Launching requested ML instances..... 2020-12-30 12:15:49 Starting - Preparing the instances for training...... 2020-12-30 12:17:12 Downloading - Downloading input data..... 2020-12-30 12:18:16 Training - Training image download completed. Training in progress..bash: cannot set terminal process group (-1): Inappropriate ioctl for device bash: no job control in this shell 2020-12-30 12:18:17,108 sagemaker-containers INFO Imported framework sagemaker pytorch container. training 2020-12-30 12:18:17,133 sagemaker pytorch container.training INFO Block until all host DNS lookup s succeed. 2020-12-30 12:18:17,137 sagemaker pytorch container.training INFO Invoking user training script. 2020-12-30 12:18:24,374 sagemaker-containers INFO Module train does not provide a setup.py. Generating setup.py Generating setup.cfg 2020-12-30 12:18:24,375 sagemaker-containers INFO Generating MANIFEST.in
Installing module with the following command: 2020-12-30 12:18:24,375 sagemaker-containers INFO 2020-12-30 12:18:24,375 sagemaker-containers INFO /usr/bin/python -m pip install -U . -r requirements.txt Processing /opt/ml/code Collecting pandas (from -r requirements.txt (line 1)) Downloading https://files.pythonhosted.org/packages/74/24/0cdbf8907e1e3bc5a8da03345c23cbed7044330bb 8f73bb12e711a640a00/pandas-0.24.2-cp35-cp35m-manylinux1 x86 64.whl (10.0MB) Collecting numpy (from -r requirements.txt (line 2)) Downloading https://files.pythonhosted.org/packages/b5/36/88723426b4ff576809fec7d73594fe17a35c27f8d 01f93637637a29ae25b/numpy-1.18.5-cp35-cp35m-manylinux1_x86 64.whl (19.9MB) Collecting nltk (from -r requirements.txt (line 3)) Downloading https://files.pythonhosted.org/packages/92/75/ce35194d8e3022203cca0d2f896dbb88689f9b3fc e8e9f9cff942913519d/nltk-3.5.zip (1.4MB) Collecting beautifulsoup4 (from -r requirements.txt (line 4)) Downloading https://files.pythonhosted.org/packages/d1/41/e6495bd7d3781cee623ce23ea6ac73282a373088f cd0ddc809a047b18eae/beautifulsoup4-4.9.3-py3-none-any.whl (115kB) Collecting html5lib (from -r requirements.txt (line 5)) Downloading https://files.pythonhosted.org/packages/6c/dd/a834df6482147d48e225a49515aabc28974ad5a4c a3215c18a882565b028/html5lib-1.1-py2.py3-none-any.whl (112kB) Requirement already satisfied, skipping upgrade: python-dateutil>=2.5.0 in /usr/local/lib/python3.5/d ist-packages (from pandas->-r requirements.txt (line 1)) (2.7.5) Collecting pytz>=2011k (from pandas->-r requirements.txt (line 1)) Downloading https://files.pythonhosted.org/packages/89/06/2c2d3034b4d6bf22f2a4ae546d16925898658a33b 4400cfb7e2c1e2871a3/pytz-2020.5-py2.py3-none-any.whl (510kB) Requirement already satisfied, skipping upgrade: click in /usr/local/lib/python3.5/dist-packages (fro m nltk->-r requirements.txt (line 3)) (7.0) Collecting joblib (from nltk->-r requirements.txt (line 3)) Downloading https://files.pythonhosted.org/packages/28/5c/cf6a2b65a321c4a209efcdf64c2689efae2cb6266 1f8f6f4bb28547cf1bf/joblib-0.14.1-py2.py3-none-any.whl (294kB) Collecting regex (from nltk->-r requirements.txt (line 3)) Downloading https://files.pythonhosted.org/packages/2e/e4/3447fed9ab29944333f48730ecff4dca92f0868c5 b188d6ab2b2078e32c2/regex-2020.11.13.tar.gz (694kB) Collecting tqdm (from nltk->-r requirements.txt (line 3)) Downloading https://files.pythonhosted.org/packages/05/bb/9403e1f30ed060e16835c9b275620ca89191a41cc c2b995b88efbc32dfd9/tqdm-4.55.0-py2.py3-none-any.whl (68kB) Collecting soupsieve>1.2; python version >= "3.0" (from beautifulsoup4->-r requirements.txt (line 4)) Downloading https://files.pythonhosted.org/packages/02/fb/1c65691a9aeb7bd6ac2aa505b84cb8b49ac29c976 411c6ab3659425e045f/soupsieve-2.1-py3-none-any.whl Requirement already satisfied, skipping upgrade: six>=1.9 in /usr/local/lib/python3.5/dist-packages (from html5lib->-r requirements.txt (line 5)) (1.11.0) Collecting webencodings (from html5lib->-r requirements.txt (line 5)) Downloading https://files.pythonhosted.org/packages/f4/24/2a3e3df732393fed8b3ebf2ec078f05546de641fe 1b667ee316ec1dcf3b7/webencodings-0.5.1-py2.py3-none-any.whl Building wheels for collected packages: nltk, train, regex Running setup.py bdist wheel for nltk: started Running setup.py bdist wheel for nltk: finished with status 'done' Stored in directory: /root/.cache/pip/wheels/ae/8c/3f/b1fe0ba04555b08b57ab52ab7f86023639a526d8bc8d3 84306 Running setup.py bdist_wheel for train: started Running setup.py bdist wheel for train: finished with status 'done' Stored in directory: /tmp/pip-ephem-wheel-cache-83tx95bn/wheels/35/24/16/37574d11bf9bde50616c67372a 334f94fa8356bc7164af8ca3 Running setup.py bdist wheel for regex: started Running setup.py bdist wheel for regex: finished with status 'done' Stored in directory: /root/.cache/pip/wheels/27/f6/66/a4243e485a0ebc73dc59033ae26c48e82526f77dbfe15 Successfully built nltk train regex Installing collected packages: numpy, pytz, pandas, joblib, regex, tqdm, nltk, soupsieve, beautifulso up4, webencodings, html5lib, train Found existing installation: numpy 1.15.4 Uninstalling numpy-1.15.4: Successfully uninstalled numpy-1.15.4 Successfully installed beautifulsoup4-4.9.3 html5lib-1.1 joblib-0.14.1 nltk-3.5 numpy-1.18.5 pandas-0.24.2 pytz-2020.5 regex-2020.11.13 soupsieve-2.1 tqdm-4.55.0 train-1.0.0 webencodings-0.5.1 You are using pip version 18.1, however version 20.3.3 is available. You should consider upgrading via the 'pip install --upgrade pip' command. 2020-12-30 12:18:47,259 sagemaker-containers INFO Invoking user script Training Env: "current host": "algo-1", "model dir": "/opt/ml/model", "num cpus": 4, "framework module": "sagemaker pytorch container.training:main", "input dir": "/opt/ml/input", "network interface name": "eth0", "user entry point": "train.py", "hyperparameters": { "hidden dim": 200, "epochs": 10 }, "channel input dirs": { "training": "/opt/ml/input/data/training" "additional framework parameters": {}, "log level": 20, "input config dir": "/opt/ml/input/config", "module_name": "train", "module dir": "s3://sagemaker-us-east-1-616376101928/sagemaker-pytorch-2020-12-30-12-14-19-049/so urce/sourcedir.tar.gz", "resource config": { "current_host": "algo-1", "network interface name": "eth0", "hosts": ["algo-1" "output dir": "/opt/ml/output", "num gpus": 1, "hosts": ["algo-1" "output intermediate dir": "/opt/ml/output/intermediate", input data config "training": { "TrainingInputMode": "File", "S3DistributionType": "FullyReplicated", "RecordWrapperType": "None" }, "job name": "sagemaker-pytorch-2020-12-30-12-14-19-049", "output data dir": "/opt/ml/output/data" Environment variables: SM NUM CPUS=4 SM NETWORK INTERFACE NAME=eth0 SM LOG LEVEL=20 PYTHONPATH=/usr/local/bin:/usr/lib/python35.zip:/usr/lib/python3.5:/usr/lib/python3.5/plat-x86 64-lin ux-gnu:/usr/lib/python3.5/lib-dynload:/usr/local/lib/python3.5/dist-packages:/usr/lib/python3/dist-pa SM FRAMEWORK PARAMS={} SM MODULE DIR=s3://sagemaker-us-east-1-616376101928/sagemaker-pytorch-2020-12-30-12-14-19-049/source/ sourcedir.tar.gz SM CHANNEL TRAINING=/opt/ml/input/data/training SM OUTPUT DIR=/opt/ml/output SM HP EPOCHS=10 SM CHANNELS=["training"] SM HPS={"epochs":10,"hidden dim":200} SM CURRENT HOST=algo-1 SM TRAINING ENV={"additional framework parameters":{},"channel input dirs":{"training":"/opt/ml/inpu t/data/training"}, "current host": "algo-1", "framework module": "sagemaker pytorch container.training:ma in", "hosts":["algo-1"], "hyperparameters":{"epochs":10, "hidden dim":200}, "input config dir":"/opt/ml/i nput/config","input data config":{"Training":{"RecordWrapperType":"None","S3DistributionType":"FullyR eplicated","TrainingInputMode":"File"}},"input dir":"/opt/ml/input","job name":"sagemaker-pytorch-202 0-12-30-12-14-19-049", "log level":20, "model dir": "/opt/ml/model", "module dir": "s3://sagemaker-us-east -1-616376101928/sagemaker-pytorch-2020-12-30-12-14-19-049/source/sourcedir.tar.gz", "module name":"tra in","network_interface_name":"eth0","num_cpus":4,"num_gpus":1,"output_data_dir":"/opt/ml/output/dat a", "output dir": "/opt/ml/output", "output intermediate dir": "/opt/ml/output/intermediate", "resource co nfig":{"current host":"algo-1", "hosts":["algo-1"], "network interface name":"eth0"}, "user entry poin t":"train.py"} SM USER ARGS=["--epochs","10","--hidden dim","200"] SM NUM GPUS=1 SM INPUT CONFIG DIR=/opt/ml/input/config SM USER ENTRY POINT=train.py SM INPUT DATA CONFIG={"training":{"RecordWrapperType":"None","S3DistributionType":"FullyReplicate d","TrainingInputMode":"File"}} SM MODULE NAME=train SM RESOURCE CONFIG={"current host":"algo-1", "hosts":["algo-1"], "network interface name":"eth0"} SM OUTPUT DATA DIR=/opt/ml/output/data SM HP HIDDEN DIM=200 SM HOSTS=["algo-1"] SM FRAMEWORK MODULE=sagemaker pytorch container.training:main SM MODEL DIR=/opt/ml/model SM OUTPUT INTERMEDIATE DIR=/opt/ml/output/intermediate SM INPUT DIR=/opt/ml/input Invoking script with the following command: /usr/bin/python -m train --epochs 10 --hidden dim 200 Using device cuda. Get train data loader. Model loaded with embedding dim 32, hidden dim 200, vocab size 5000. Epoch: 1, BCELoss: 0.6742265297442066 Epoch: 2, BCELoss: 0.628087999869366 Epoch: 3, BCELoss: 0.5520178152590381 Epoch: 4, BCELoss: 0.4870846721590782 Epoch: 5, BCELoss: 0.4232300957854913 Epoch: 6, BCELoss: 0.3760740416390555 Epoch: 7, BCELoss: 0.3557087152588124 Epoch: 8, BCELoss: 0.32314337942065025 Epoch: 9, BCELoss: 0.3091213618006025 2020-12-30 12:21:49 Uploading - Uploading generated training modelEpoch: 10, BCELoss: 0.2999043707944 2020-12-30 12:21:46,312 sagemaker-containers INFO Reporting training SUCCESS 2020-12-30 12:21:56 Completed - Training job completed Training seconds: 284 Billable seconds: 284 Step 5: Testing the model As mentioned at the top of this notebook, we will be testing this model by first deploying it and then sending the testing data to the deployed endpoint. We will do this so that we can make sure that the deployed model is working correctly. Step 6: Deploy the model for testing Now that we have trained our model, we would like to test it to see how it performs. Currently our model takes input of the form review length, review[500] where review[500] is a sequence of 500 integers which describe the words present in the review, encoded using word dict. Fortunately for us, SageMaker provides built-in inference code for models with simple inputs such as this. There is one thing that we need to provide, however, and that is a function which loads the saved model. This function must be called model fn() and takes as its only parameter a path to the directory where the model artifacts are stored. This function must also be present in the python file which we specified as the entry point. In our case the model loading function has been provided and so no changes need to be made. NOTE: When the built-in inference code is run it must import the <code>model_fn()</code> method from the <code>train.py</code> file. This is why the training code is wrapped in a main guard (ie, if name == ' main ':) Since we don't need to change anything in the code that was uploaded during training, we can simply deploy the current model as-is. **NOTE:** When deploying a model you are asking SageMaker to launch an compute instance that will wait for data to be sent to it. As a result, this compute instance will continue to run until you shut it down. This is important to know since the cost of a deployed endpoint depends on how long it has been running for. In other words If you are no longer using a deployed endpoint, shut it down! **TODO:** Deploy the trained model. In [30]: # TODO: Deploy the trained model predictor = estimator.deploy(initial instance count = 1, instance type = 'ml.m4.xlarge') Parameter image will be renamed to image_uri in SageMaker Python SDK v2. create_image_uri' will be deprecated in favor of 'ImageURIProvider' class in SageMaker Python SDK v' -----! Step 7 - Use the model for testing Once deployed, we can read in the test data and send it off to our deployed model to get some results. Once we collect all of the results we can determine how accurate our model is. In [31]: test X = pd.concat([pd.DataFrame(test X len), pd.DataFrame(test X)], axis=1) In [32]: # We split the data into chunks and send each chunk seperately, accumulating the results. def predict(data, rows=512): split_array = np.array_split(data, int(data.shape[0] / float(rows) + 1)) predictions = np.array([]) for array in split array: predictions = np.append(predictions, predictor.predict(array)) return predictions In [33]: predictions = predict(test X.values) predictions = [round(num) for num in predictions] In [34]: **from sklearn.metrics import** accuracy_score accuracy_score(test_y, predictions) Out[34]: 0.85044 Question: How does this model compare to the XGBoost model you created earlier? Why might these two models perform differently on this dataset? Which do you think is better for sentiment analysis? Answer: First of all just to outline the difference, Recurrent neural networks (RNN) are essentially just a family of neural networks for processing sequential data. RNNs consider and uses previous information from the input sequence for the prediction. It is typically used for Sequence Classification, Sequence Labelling, Sequence Generation. On the other hand, XGBoost is a decision-tree-based ensemble Machine Learning algorithm that uses a gradient boosting framework where its commonly used for prediction problems involving unstructured data (images, text, etc.). Recall that in our lectures, specifically the Mini-Project example we went through the accuracy was approximately around 86%, on the other hand, while using the LSTM Classifier above we have around ~85%. In terms of accuracy, it's not extremely worst compare to our XGBoost model, it's highly likely with hyperparameter tuning we can further increase the accuracy. It performed a bit differently because in the lectures we approached it using BOW (Bag of words). It seems like if we spend more time on our RNN architecture in terms of tuning it can perform better and could be better for sentiment analysis. (TODO) More testing We now have a trained model which has been deployed and which we can send processed reviews to and which returns the predicted sentiment. However, ultimately we would like to be able to send our model an unprocessed review. That is, we would like to send the review itself as a string. For example, suppose we wish to send the following review to our model. In [35]: test review = 'The simplest pleasures in life are the best, and this film is one of them. Combining a r ather basic storyline of love and adventure this movie transcends the usual weekend fair with wit and u nmitigated charm.' The question we now need to answer is, how do we send this review to our model? Recall in the first section of this notebook we did a bunch of data processing to the IMDb dataset. In particular, we did two specific things to the provided reviews. • Removed any html tags and stemmed the input • Encoded the review as a sequence of integers using word dict In order process the review we will need to repeat these two steps. TODO: Using the review to words and convert and pad methods from section one, convert test review into a numpy array test data suitable to send to our model. Remember that our model expects input of the form review length, review [500]. In [36]: #tes = review to words(test review) #data_X, data_len = convert_and_pad(word_dict, tes) #data len Fix suggested by reviewer, as I needed to concatenate the length to the test_data In [37]: # TODO: Convert test review into a form usable by the model and save the results in test_data test data, leng = convert and pad(word dict, review to words(test review)) len test data = np.array([np.array([leng] + test data)]) Now that we have processed the review, we can send the resulting array to our model to predict the sentiment of the review. In [38]: predictor.predict(len test data) Out[38]: array(0.7142381, dtype=float32) Since the return value of our model is close to 1, we can be certain that the review we submitted is positive. Delete the endpoint Of course, just like in the XGBoost notebook, once we've deployed an endpoint it continues to run until we tell it to shut down. Since we are done using our endpoint for now, we can delete it. In [39]: estimator.delete endpoint() estimator.delete endpoint() will be deprecated in SageMaker Python SDK v2. Please use the delete endp oint() function on your predictor instead. Step 6 (again) - Deploy the model for the web app Now that we know that our model is working, it's time to create some custom inference code so that we can send the model a review which has not been processed and have it determine the sentiment of the review. As we saw above, by default the estimator which we created, when deployed, will use the entry script and directory which we provided when creating the model. However, since we now wish to accept a string as input and our model expects a processed review, we need to write some custom inference code. We will store the code that we write in the serve directory. Provided in this directory is the model.py file that we used to construct our model, a utils.py file which contains the review to words and convert and pad pre-processing functions which we used during the initial data processing, and predict.py, the file which will contain our custom inference code. Note also that requirements.txt is present which will tell SageMaker what Python libraries are required by our custom inference code. When deploying a PyTorch model in SageMaker, you are expected to provide four functions which the SageMaker inference container will use. • model fn: This function is the same function that we used in the training script and it tells SageMaker how to load our model. • input fn: This function receives the raw serialized input that has been sent to the model's endpoint and its job is to de-serialize and make the input available for the inference code. • output fn: This function takes the output of the inference code and its job is to serialize this output and return it to the caller of the model's endpoint. • predict fn: The heart of the inference script, this is where the actual prediction is done and is the function which you will need to complete. For the simple website that we are constructing during this project, the <code>input_fn</code> and <code>output_fn</code> methods are relatively straightforward. We only require being able to accept a string as input and we expect to return a single value as output. You might imagine though that in a more complex application the input or output may be image data or some other binary data which would require some effort to serialize. (TODO) Writing inference code Before writing our custom inference code, we will begin by taking a look at the code which has been provided. In [44]: !pygmentize serve/predict.py import argparse import json import os import pickle import sys import sagemaker containers import pandas as pd import <u>numpy</u> as <u>np</u> import torch import torch.nn as nn import torch.optim as optim import torch.utils.data from model import LSTMClassifier from utils import review_to_words, convert_and_pad def model fn(model dir): """Load the PyTorch model from the `model dir` directory.""" print("Loading model.") # First, load the parameters used to create the model. model info = {} model_info_path = os.path.join(model_dir, 'model_info.pth') with open(model_info_path, 'rb') as f: model info = torch.load(f) print("model_info: {}".format(model_info)) # Determine the device and construct the model. device = torch.device("cuda" if torch.cuda.is_available() else "cpu") model = LSTMClassifier(model_info['embedding_dim'], model_info['hidden_dim'], model_info['vocab_s ize']) # Load the store model parameters. model_path = os.path.join(model_dir, 'model.pth') with open(model_path, 'rb') as f: model.load_state_dict(torch.load(f)) # Load the saved word_dict. word_dict_path = os.path.join(model_dir, 'word_dict.pkl') with open(word_dict_path, 'rb') as f: model.word_dict = pickle.load(f) model.to(device).eval() print("Done loading model.") return model def input_fn(serialized_input_data, content_type): print('Deserializing the input data.') if content_type == 'text/plain': data = serialized_input_data.decode('utf-8') return data raise Exception('Requested unsupported ContentType in content_type: ' + content_type) def output_fn(prediction_output, accept): print('Serializing the generated output.') return str(prediction_output) def predict_fn(input_data, model): print('Inferring sentiment of input data.') device = torch.device("cuda" if torch.cuda.is_available() else "cpu") if model.word dict is None: raise Exception ('Model has not been loaded properly, no word dict.') # TODO: Process input data so that it is ready to be sent to our model. You should produce two variables: data_X - A sequence of length 500 which represents the converted review data len - The length of the review words = review to words(input data) data_X, data_len = convert_and_pad(model.word_dict, words) # Using data_X and data_len we construct an appropriate input tensor. Remember # that our model expects input data of the form 'len, review[500]'. data pack = np.hstack((data len, data X)) data_pack = data_pack.reshape(1, -1) data = torch.from numpy(data pack) data = data.to(device) # Make sure to put the model into evaluation mode model.eval() # TODO: Compute the result of applying the model to the input data. The variable `result` should be a numpy array which contains a single integer which is either 1 or 0 with torch.no grad(): result = model.forward(data) #Round off then apply int function result = int(np.round(result.numpy())) return result As mentioned earlier, the model fn method is the same as the one provided in the training code and the input fn and output fn methods are very simple and your task will be to complete the predict fn method. Make sure that you save the completed file as predict.py in the serve directory. **TODO**: Complete the predict fn() method in the serve/predict.py file. Deploying the model Now that the custom inference code has been written, we will create and deploy our model. To begin with, we need to construct a new PyTorchModel object which points to the model artifacts created during training and also points to the inference code that we wish to use. Then we can call the deploy method to launch the deployment container. NOTE: The default behaviour for a deployed PyTorch model is to assume that any input passed to the predictor is a numpy array. In our case we want to send a string so we need to construct a simple wrapper around the RealTimePredictor class to accomodate simple strings. In a more complicated situation you may want to provide a serialization object, for example if you wanted to sent image data. In [45]: from sagemaker.predictor import RealTimePredictor from sagemaker.pytorch import PyTorchModel class StringPredictor(RealTimePredictor): def __init__(self, endpoint_name, sagemaker_session): super(StringPredictor, self).__init__(endpoint_name, sagemaker_session, content_type='text/plai n') model = PyTorchModel(model data=estimator.model data, role = role, framework version='0.4.0', entry point='predict.py', source dir='serve', predictor_cls=StringPredictor) predictor = model.deploy(initial instance count=1, instance type='ml.m4.xlarge') Parameter image will be renamed to image_uri in SageMaker Python SDK v2. 'create_image_uri' will be deprecated in favor of 'ImageURIProvider' class in SageMaker Python SDK v 2. -----! Testing the model Now that we have deployed our model with the custom inference code, we should test to see if everything is working. Here we test our model by loading the first 250 positive and negative reviews and send them to the endpoint, then collect the results. The reason for only sending some of the data is that the amount of time it takes for our model to process the input and then perform inference is quite long and so testing the entire data set would be prohibitive. In [46]: import glob def test reviews(data dir='../data/aclImdb', stop=250): results = [] ground = [] # We make sure to test both positive and negative reviews for sentiment in ['pos', 'neg']: path = os.path.join(data dir, 'test', sentiment, '*.txt') files = glob.glob(path) files_read = 0 print('Starting ', sentiment, ' files') # Iterate through the files and send them to the predictor for f in files: with open(f) as review: # First, we store the ground truth (was the review positive or negative) if sentiment == 'pos': ground.append(1) ground.append(0) # Read in the review and convert to 'utf-8' for transmission via HTTP review input = review.read().encode('utf-8') # Send the review to the predictor and store the results results.append(float(predictor.predict(review input))) # Sending reviews to our endpoint one at a time takes a while so we # only send a small number of reviews files read += 1 if files_read == stop: break return ground, results In [47]: ground, results = test_reviews() Starting pos files Starting neg files In [48]: from sklearn.metrics import accuracy score accuracy_score(ground, results) Out[48]: 0.842 As an additional test, we can try sending the test review that we looked at earlier. In [49]: predictor.predict(test review) Out[49]: b'1' Now that we know our endpoint is working as expected, we can set up the web page that will interact with it. If you don't have time to finish the project now, make sure to skip down to the end of this notebook and shut down your endpoint. You can deploy it again when you come back. Step 7 (again): Use the model for the web app **TODO:** This entire section and the next contain tasks for you to complete, mostly using the AWS console. So far we have been accessing our model endpoint by constructing a predictor object which uses the endpoint and then just using the predictor object to perform inference. What if we wanted to create a web app which accessed our model? The way things are set up currently makes that not possible since in order to access a SageMaker endpoint the app would first have to authenticate with AWS using an IAM role which included access to SageMaker endpoints. However, there is an easier way! We just need to use some additional AWS services. 📝The diagram above gives an overview of how the various services will work together. On the far right is the model which we trained above and which is deployed using SageMaker. On the far left is our web app that collects a user's movie review, sends it off and expects a positive or negative sentiment in return. In the middle is where some of the magic happens. We will construct a Lambda function, which you can think of as a straightforward Python function that can be executed whenever a specified event occurs. We will give this function permission to send and recieve data from a SageMaker endpoint. Lastly, the method we will use to execute the Lambda function is a new endpoint that we will create using API Gateway. This endpoint will be a url that listens for data to be sent to it. Once it gets some data it will pass that data on to the Lambda function and then return whatever the Lambda function returns. Essentially it will act as an interface that lets our web app communicate with the Lambda function. Setting up a Lambda function The first thing we are going to do is set up a Lambda function. This Lambda function will be executed whenever our public API has data sent to it. When it is executed it will receive the data, perform any sort of processing that is required, send the data (the review) to the SageMaker endpoint we've created and then return the result. Part A: Create an IAM Role for the Lambda function Since we want the Lambda function to call a SageMaker endpoint, we need to make sure that it has permission to do so. To do this, we will construct a role that we can later give the Lambda function. Using the AWS Console, navigate to the IAM page and click on Roles. Then, click on Create role. Make sure that the AWS service is the type of trusted entity selected and choose Lambda as the service that will use this role, then click Next: Permissions. In the search box type sagemaker and select the check box next to the AmazonSageMakerFullAccess policy. Then, click on Next: Review. Lastly, give this role a name. Make sure you use a name that you will remember later on, for example LambdaSageMakerRole. Then, click on Create role. Part B: Create a Lambda function Now it is time to actually create the Lambda function. Using the AWS Console, navigate to the AWS Lambda page and click on Create a function. When you get to the next page, make sure that Author from scratch is selected. Now, name your Lambda function, using a name that you will remember later on, for example sentiment analysis func . Make sure that the Python 3.6 runtime is selected and then choose the role that you created in the previous part. Then, click on Create Function. On the next page you will see some information about the Lambda function you've just created. If you scroll down you should see an editor in which you can write the code that will be executed when your Lambda function is triggered. In our example, we will use the code below. # We need to use the low-level library to interact with SageMaker since the SageMaker API # is not available natively through Lambda. import boto3 def lambda handler(event, context): # The SageMaker runtime is what allows us to invoke the endpoint that we've created. runtime = boto3.Session().client('sagemaker-runtime') # Now we use the SageMaker runtime to invoke our endpoint, sending the review we were given response = runtime.invoke endpoint(EndpointName = '**ENDPOINT NAME HERE**', # The name of the endpoint we created ContentType = 'text/plain', # The data fo rmat that is expected Body = event['body']) # The actual review # The response is an HTTP response whose body contains the result of our inference result = response['Body'].read().decode('utf-8') return { 'statusCode' : 200, 'headers' : { 'Content-Type' : 'text/plain', 'Access-Control-Allow-Origin' : '*' }, 'body' : result } Once you have copy and pasted the code above into the Lambda code editor, replace the **ENDPOINT NAME HERE** portion with the name of the endpoint that we deployed earlier. You can determine the name of the endpoint using the code cell below. In [50]: predictor.endpoint Out[50]: 'sagemaker-pytorch-2020-12-30-13-04-03-574' Once you have added the endpoint name to the Lambda function, click on Save. Your Lambda function is now up and running. Next we need to create a way for our web app to execute the Lambda function. Setting up API Gateway Now that our Lambda function is set up, it is time to create a new API using API Gateway that will trigger the Lambda function we have just created. Using AWS Console, navigate to Amazon API Gateway and then click on Get started. On the next page, make sure that New API is selected and give the new api a name, for example, sentiment analysis api. Then, click on Create API. Now we have created an API, however it doesn't currently do anything. What we want it to do is to trigger the Lambda function that we created earlier. Select the Actions dropdown menu and click Create Method. A new blank method will be created, select its dropdown menu and select POST, then click on the check mark beside it. For the integration point, make sure that Lambda Function is selected and click on the Use Lambda Proxy integration. This option makes sure that the data that is sent to the API is then sent directly to the Lambda function with no processing. It also means that the return value must be a proper response object as it will also not be processed by API Gateway. Type the name of the Lambda function you created earlier into the Lambda Function text entry box and then click on Save. Click on OK in the pop-up box that then appears, giving permission to API Gateway to invoke the Lambda function you created. The last step in creating the API Gateway is to select the Actions dropdown and click on Deploy API. You will need to create a new Deployment stage and name it anything you like, for example prod. You have now successfully set up a public API to access your SageMaker model. Make sure to copy or write down the URL provided to invoke your newly created public API as this will be needed in the next step. This URL can be found at the top of the page, highlighted in blue next to the text Invoke URL. Step 4: Deploying our web app Now that we have a publicly available API, we can start using it in a web app. For our purposes, we have provided a simple static html file which can make use of the public api you created earlier. In the website folder there should be a file called index.html . Download the file to your computer and open that file up in a text editor of your choice. There should be a line which contains **REPLACE WITH PUBLIC API URL**. Replace this string with the url that you wrote down in the last step and then save the file. Now, if you open index.html on your local computer, your browser will behave as a local web server and you can use the provided site to interact with your SageMaker model. If you'd like to go further, you can host this html file anywhere you'd like, for example using github or hosting a static site on Amazon's S3. Once you have done this you can share the link with anyone you'd like and have them play with it too! Important Note In order for the web app to communicate with the SageMaker endpoint, the endpoint has to actually be deployed and running. This means that you are paying for it. Make sure that the endpoint is running when you want to use the web app but that you shut it down when you don't need it, otherwise you will end up with a surprisingly large AWS bill. **TODO:** Make sure that you include the edited <code>index.html</code> file in your project submission. Now that your web app is working, trying playing around with it and see how well it works. Question: Give an example of a review that you entered into your web app. What was the predicted sentiment of your example review? Answer: Review given: "Such a lovely movie really wish i could watch it again with my family!" Sentiment Prediction: Your review was POSITIVE! Delete the endpoint Remember to always shut down your endpoint if you are no longer using it. You are charged for the length of time that the endpoint is running so if you forget and leave it on you could end up with an unexpectedly large bill. In [51]: predictor.delete endpoint() In []: