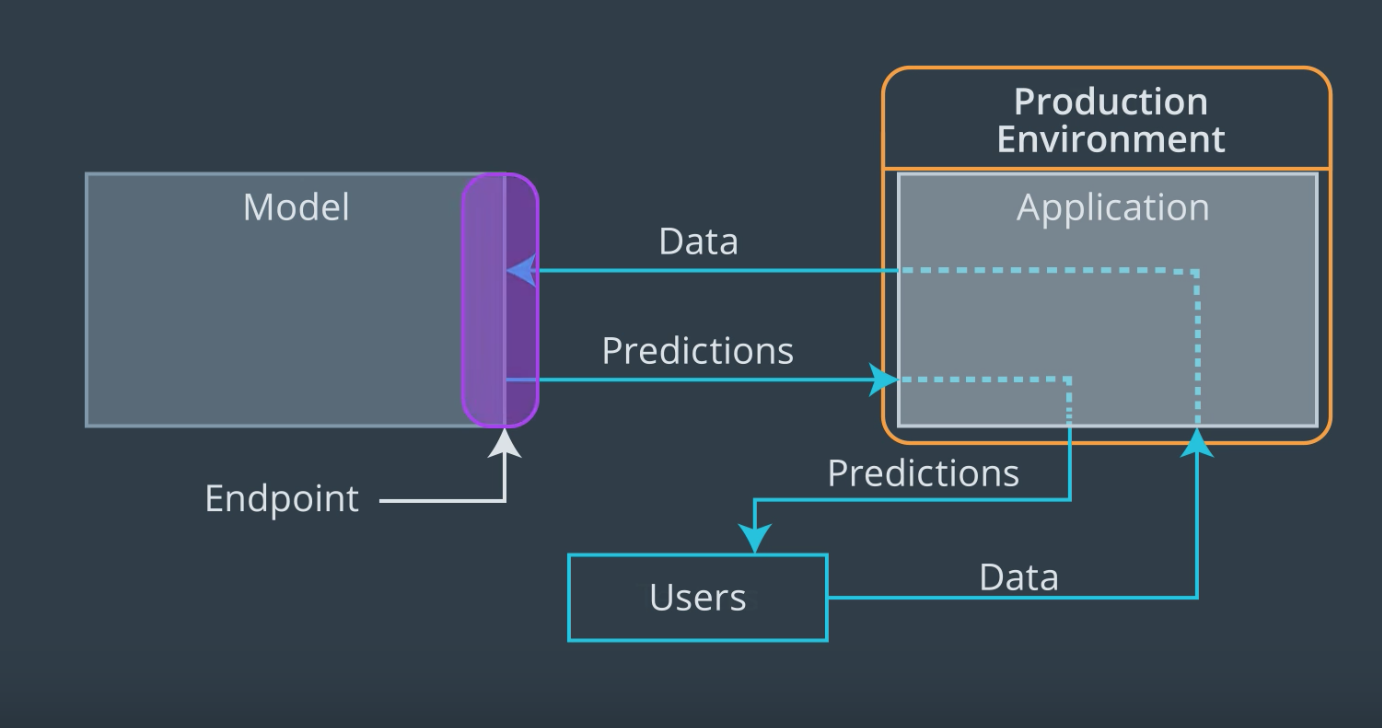
**Deploying a Model in SageMaker**

**Deploying a Model in SageMaker**

In this lesson, we're going to take a look at how we can use a model that has been created in SageMaker. We will do this by first *deploying* our model. For us, this means using SageMaker's functionality to **create an endpoint that will be used as a way to send data to our model**.

*Recall, from the first lesson in this section, that an endpoint is basically a way to allow a model and an application to communicate. An application, such as a web app, will be responsible for accepting user input data, and through an endpoint we can send that data to our model, which will produce predictions that can be sent back to our application!*



For our purposes **an endpoint is just a URL**. Instead of returning a web page, like a typical url, this endpoint URL returns the *results* of performing inference. In addition, we are able to send data to this URL so that our model knows what to perform inference on. We won't go too far into the details of how this is all set up since SageMaker does most of the heavy lifting for us.

An important aspect that we will encounter is that SageMaker endpoints are secured. In this case, that means that only other AWS services with permission to access SageMaker endpoints can do so.

To start with, we won't need to worry about this too much since we will be working inside of a SageMaker notebook and so we will be able to access our deployed endpoints easily.

Later on we will talk about how to set things up so that a simple web app, which doesn't need to be given special permission, can access our SageMaker endpoint.

# Boston Housing Example

Now that you've had some time to try and build models using SageMaker, we are going to learn how to deploy those models so that our models can be interacted with using an endpoint.

Inside of the Tutorials folder is the Boston Housing - XGBoost (Deploy) - High Level.ipynb notebook which we will be looking at in the video below.

Using the high level approach makes deploying a trained model pretty straightforward. All we need to do is call the deploy method and SageMaker takes care of the rest.

Similarly, sending data to the deployed endpoint and capturing the resulting inference is easy too as SageMaker wraps everything up into the predict method, provided we make sure that our data is serialized correctly. In our case, serializing means converting the data structure we wish to send to our endpoint into a string, something that can be transferred using HTTP.

In the next video we'll take a more in-depth look at how our model is being deployed.

## WARNING - SHUT DOWN YOUR DEPLOYED ENDPOINT

Sorry for the yelling, but this is pretty important. The cost of a deployed endpoint is based on the length of time that it is running. This means that if you aren't using an endpoint you **really** need to shut it down.

# Boston Housing In-Depth

Now we will look at deploying a model using the low level approach. This method requires us to describe the various properties that our endpoint should have and what inference code and model should be used.

To follow along, open up the Boston Housing - XGBoost (Deploy) - Low Level.ipynb notebook in the Tutorials folder.

Using the low level approach to deploy our model requires us to create an endpoint, which will be used to send data to our model and to get inference results.

In order to create an endpoint in SageMaker, we first need to describe an endpoint configuration. This describes to SageMaker the various properties we want our endpoint to have. Once we've created the endpoint configuration we can ask SageMaker to create an endpoint with the properties we want.

The actual endpoint that is created by SageMaker is a combination of a compute instance (some remote server) running a docker container with the inference code on it and a URL that data can be sent to and returned from. This URL is used as an interface to the compute instance, which receives data, performs inference using our model and returns the result.

# Deploying and Using a Sentiment Analysis Model

You've learned how to create and train models in SageMaker and how you can deploy them. In this example we are going to look at how we can make use of a deployed model in a simple web app.

In order for our simple web app to interact with the deployed model we are going to have to solve a couple problems.

The first obstacle is something that has been mentioned earlier.

*The endpoint that is created when we deploy a model using SageMaker is secured, meaning that only entities that are authenticated with AWS can send or receive data from the deployed model. This is a problem since authenticating for the purposes of a simple web app is a bit more work than we'd like.*

So we will need to find a way to work around this.

*The second obstacle is that our deployed model expects us to send it a review after it has been processed. That is, it assumes we have already tokenized the review and then created a bag of words encoding. However, we want our user to be able to type any review into our web app.*

We will also see how we can overcome this.

To solve these issues we are going to need to use some additional Amazon services. In particular, we are going to look at Amazon Lambda and API Gateway.

In the mean time, I would encourage you to take a look at the IMDB Sentiment Analysis - XGBoost - Web App.ipynb notebook in the Tutorials folder. In the coming videos we will go through this notebook in detail, however, each of the steps involved is pretty well documented in the notebook itself.

## Text Processing

I mentioned that one of our tasks will be to convert any user input text into data that our deployed model can see as input. You've seen a few examples of text pre-processing and the steps usually go something like this:

1. Get rid of any special characters like punctuation
2. Convert all text to lowercase and split into individual words
3. Create a vocabulary that assigns each unique word a numerical value or converts words into a vector of numbers

This last step is often called **word tokenization** or vectorization.

And in the next example, you'll see exactly how I do these processing steps; I'll also be vectorizing words using a method called **bag of words**. If you'd like to learn more about bag of words, please check out the video below, recorded by another of our instructors, Arpan!

## Bag of Words

You can read more about the bag of words model, and its applications, [on this page](https://en.wikipedia.org/wiki/Bag-of-words_model). It's a useful way to represent words based on their frequency of occurrence in a text.

# Building and Deploying the Model

*Remember to shutdown your endpoint, if you are not going to use it for a while; we'll remind you again at the end of this video.*

To begin with, we are going to extend the mini-project that you worked on in the last lesson by deploying it. There are a couple of changes made to the way that data is processed in this version and the reason for this is to simplify some of what follows.

For the most part, however, we simply add on an extra deployment step to the sentiment analysis mini-project and then test that our deployed endpoint is working correctly.

Once this is done we know that we have a sentiment analysis model that has been trained, is performing well and is working, a great place to start!

# Don't forget to SHUT DOWN your endpoint!

**How to use a Deployed Model**

As mentioned earlier, there are two obstacles we are going to need to overcome. The first is the security issue and the second is data processing. The way that we are going to approach solving these issues is by making use of Amazon Lambda and API Gateway.

The structure for our web app will look like the diagram below.

What this means is that when someone uses our web app, the following will occur.

1. To begin with, a user will type out a review and enter it into our web app.
2. Then, our web app will send that review to an endpoint that we created using API Gateway. This endpoint will be constructed so that anyone (including our web app) can use it.
3. API Gateway will forward the data on to the Lambda function
4. Once the Lambda function receives the user's review, it will process that review by tokenizing it and then creating a bag of words encoding of the result. After that, it will send the processed review off to our deployed model.
5. Once the deployed model performs inference on the processed review, the resulting sentiment will be returned back to the Lambda function.
6. Our Lambda function will then return the sentiment result back to our web app using the endpoint that was constructed using API Gateway.

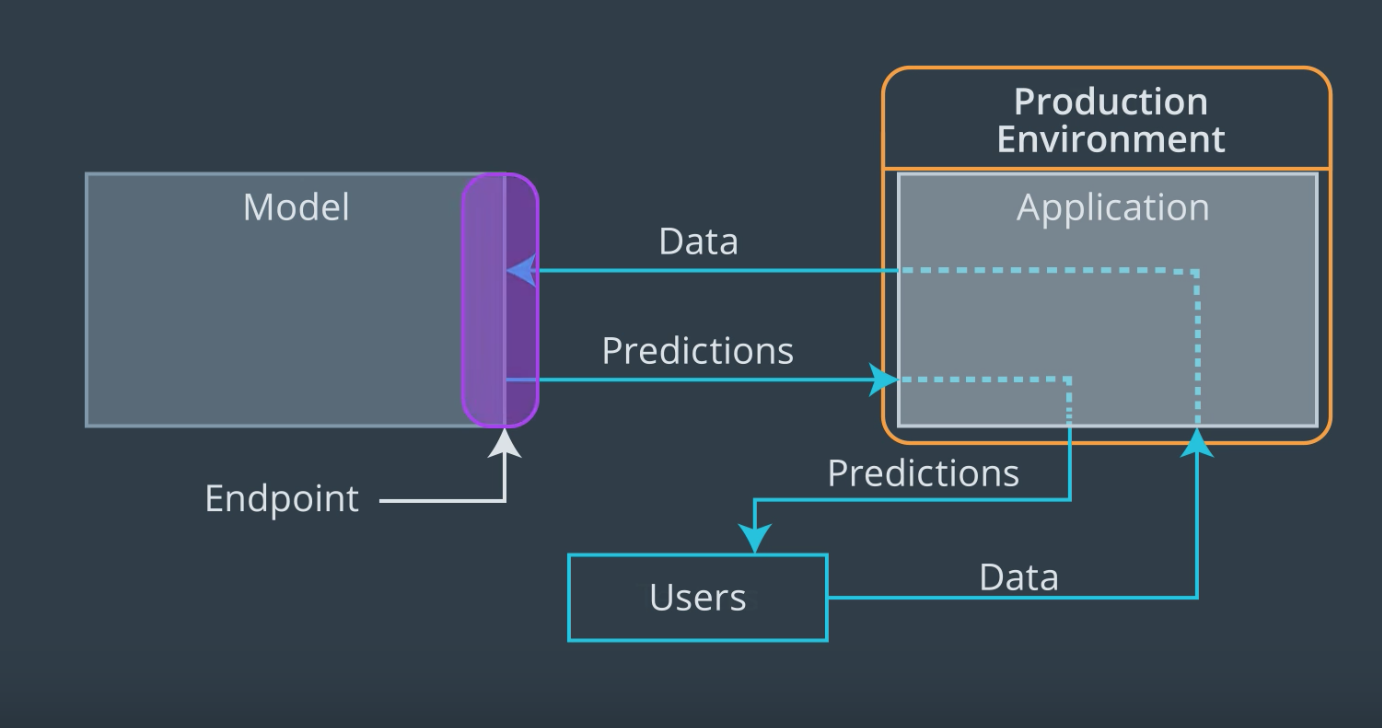
**Don't forget!**

Currently our endpoint is running. The reason for this is that in the next few videos we are going to interact with our deployed endpoint. If you are following along, don't forget that your endpoint is running. If you need to take a break, don't forget to shut down your endpoint!

## Creating and Using Endpoints

You've just learned a lot about how to use SageMaker to deploy a model and perform inference on some data. Now is a good time to review some of the key steps that we've covered. You have experience processing data and creating estimators/models, so I'll focus on what you've learned about endpoints.

An endpoint, in this case, is a URL that allows an application and a model to speak to one another.



### Endpoint steps

* You can start an endpoint by calling .deploy() on an estimator and passing in some information about the instance.

xgb\_predictor = xgb.deploy(initial\_instance\_count = 1, instance\_type = 'ml.m4.xlarge')

* Then, you need to tell your endpoint, what type of data it expects to see as input (like .csv).

**from** sagemaker.predictor **import** csv\_serializer

xgb\_predictor.content\_type = 'text/csv'

xgb\_predictor.serializer = csv\_serializer

* Then, perform inference; you can pass some data as the "Body" of a message, to an endpoint and get a response back!

response = runtime.invoke\_endpoint(EndpointName = xgb\_predictor.endpoint, *# The name of the endpoint we created*

ContentType = 'text/csv', *# The data format that is expected*

Body = ','.join([str(val) **for** val **in** test\_bow]).encode('utf-8'))

* The inference data is stored in the "Body" of the response, and can be retrieved:
* response = response['Body'].read().decode('utf-8')
* print(response)
* Finally, do not forget to **shut down your endpoint** when you are done using it.

xgb\_predictor.delete\_endpoint()

**Building a Lambda Function**

In general, a Lambda function is an example of a 'Function as a Service'. It lets you perform actions in response to certain events, called triggers. Essentially, you get to describe some events that you care about, and when those events occur, your code is executed.

For example, you could set up a trigger so that whenever data is uploaded to a particular S3 bucket, a Lambda function is executed to process that data and insert it into a database somewhere.

One of the big advantages to Lambda functions is that since the amount of code that can be contained in a Lambda function is relatively small, you are only charged for the *number* of executions.

In our case, the Lambda function we are creating is meant to process user input and interact with our deployed model. Also, the trigger that we will be using is the endpoint that we will create using API Gateway.

**Create a Lambda Function**

The steps to create a lambda function are outlined in the notebook and here, for convenience.

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.

Task List

* 

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. Remember from earlier that in order to process the user provided input and send it to our endpoint we need to gather two pieces of information:

1. The name of the endpoint, and
2. the vocabulary object.

We will copy these pieces of information to our Lambda function, after we create it.

Task List

* 

To start, 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\_xgboost\_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... you should copy and past the code below.

* 

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 will also need to copy the vocabulary dict to the appropriate place in the code at the beginning of the lambda\_handler method.

* 

Once you have added the endpoint name to the Lambda function, click on Save. Your Lambda function is now up and running!

*# We need to use the low-level library to interact with SageMaker since the SageMaker API*

*# is not available natively through Lambda.*

**import** boto3

*# And we need the regular expression library to do some of the data processing*

**import** re

REPLACE\_NO\_SPACE = re.compile("(\.)|(\;)|(\:)|(\!)|(\')|(\?)|(\,)|(\")|(\()|(\))|(\[)|(\])")

REPLACE\_WITH\_SPACE = re.compile("(<br\s\*/><br\s\*/>)|(\-)|(\/)")

**def** **review\_to\_words**(review):

words = REPLACE\_NO\_SPACE.sub("", review.lower())

words = REPLACE\_WITH\_SPACE.sub(" ", words)

**return** words

**def** **bow\_encoding**(words, vocabulary):

bow = [0] \* len(vocabulary) *# Start by setting the count for each word in the vocabulary to zero.*

**for** word **in** words.split(): *# For each word in the string*

**if** word **in** vocabulary: *# If the word is one that occurs in the vocabulary, increase its count.*

bow[vocabulary[word]] += 1

**return** bow

**def** **lambda\_handler**(event, context):

vocab = "\*\*\* ACTUAL VOCABULARY GOES HERE \*\*\*"

words = review\_to\_words(event['body'])

bow = bow\_encoding(words, vocab)

*# 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/csv', *# The data format that is expected*

Body = ','.join([str(val) **for** val **in** bow]).encode('utf-8')) *# The actual review*

*# The response is an HTTP response whose body contains the result of our inference*

result = response['Body'].read().decode('utf-8')

*# Round the result so that our web app only gets '1' or '0' as a response.*

result = round(float(result))

**return** {

'statusCode' : 200,

'headers' : { 'Content-Type' : 'text/plain', 'Access-Control-Allow-Origin' : '\*' },

'body' : str(result)

}

# Building an API

At this point we've created and deployed a model, and we've constructed a Lambda function that can take care of processing user data, sending it off to our deployed model and returning the result. What we need to do now is set up some way to send our user data to the Lambda function.

The way that we will do this is using a service called API Gateway. Essentially, API Gateway allows us to create an HTTP endpoint (a web address). In addition, we can set up what we want to happen when someone tries to send data to our constructed endpoint.

In our application, we want to set it up so that when data is sent to our endpoint, we trigger the Lambda function that we created earlier, making sure to send the data to our Lambda function for processing. Then, once the Lambda function has retrieved the inference results from our model, we return the results back to the original caller.

# Using the Final Web Application

Now we get to reap the rewards of all our hard work, we get to deploy our web app!

The back end of our app has been set up so at this point all we need to do is finish up the user facing portion, the website itself. To do this we just need to tell our website where it should send data to.

# Don't forget!

In order for our web app to work, we need to have our model deployed. This means that we are incurring a cost. So, once you have finished playing with your newly created web app, make sure to shut it down!

You may also want to clean up the endpoint that you constructed and the Lambda function. This isn't too important, however, since each of these services only incur a cost when used.

### Some notes on Lambda and Gateway usage

For Lambda functions you are only charged per execution, which for this class will be very few and still within the free tier. Deleting a lambda function is just a good cleanup step; you won't be charged if you just leave it there (without executing it). Similarly, for APIs created using API Gateway you are only charged per request, and the number of requests we require in this course should still fall under the free tier.

# What have we learned so far?

In this lesson we learned how to deploy a model that has been created using SageMaker. We took a look at how to construct endpoints and how to use those endpoints to send data to a deployed model.

In addition, we looked at what we needed to do if we wanted anyone to have access to our deployed model. To make this work we first implemented a Lambda function that took care of data processing and interacting with the model. Then we created an interface through which we could send data to our Lambda function using API Gateway.

## How does the data flow?

Describe how the data flows through our sentiment analysis web app.

### Your reflection

**.**

### Things to think about

To begin with, the user enters a review on our website.

Next, our website sends that data off to an endpoint, created using API Gateway.

Our endpoint acts as an interface to our Lambda function so our user data gets sent to the Lambda function.

Our Lambda function processes the user data and sends it off to the deployed model's endpoint.

The deployed model perform inference on the processed data and returns the inference results to the Lambda function.

The Lambda function returns the results to the original caller using the endpoint constructed using API Gateway.

Lastly, the website receives the inference results and displays those results to the user.

# What's next?

In the next lesson we are going to look at hyperparameter tuning. This is a method by which we can test a variety of different hyperparameters and chose the ones that work best for our data set.

While doing this we will also take a look at CloudWatch, which is a service that allows us to look at the logs generated by the various SageMaker tasks we perform.