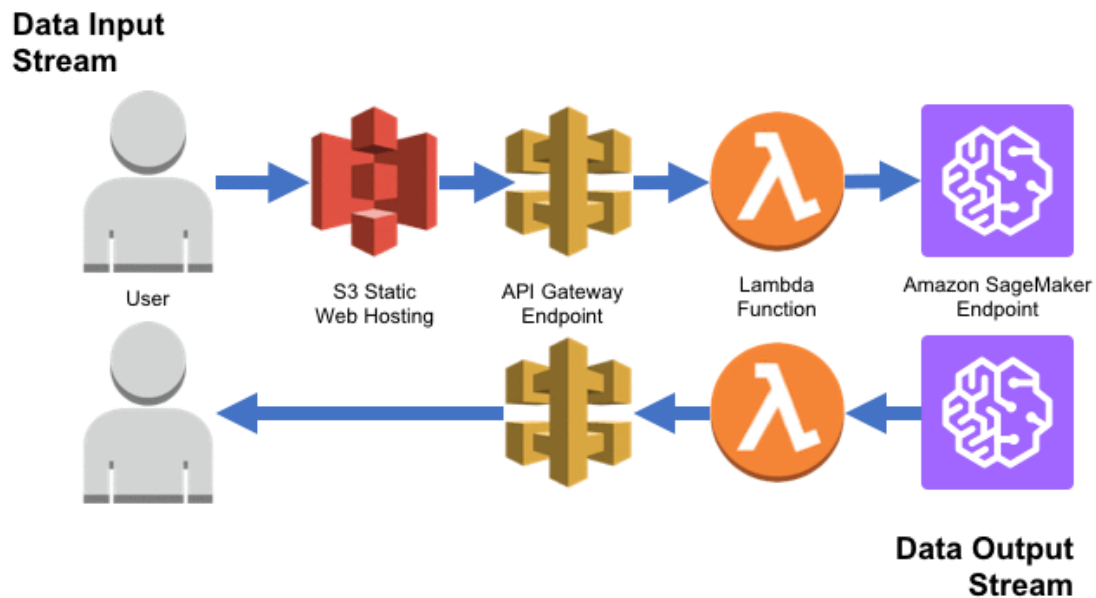


General Overview:

I will be deploying my trained CNN (Keras model) in AWS as a *static web application*. Users will be able to input emails - either spam or legitimate - in the web application, and the application will return the predictions back to the user in a readable format.

Deployment Architecture:



Specific tools used in my system:

- *Amazon API Gateway* for serverless API hosting,
- *Amazon S3* for hosting the web application,
- *Amazon SageMaker* for model deployment,
- *AWS Lambda* to connect my SageMaker endpoint to my front-end web app (and do some input/output processing), and
- *AWS X-Ray* and *Amazon CloudWatch* to analyze and debug

Inputs and Outputs:

For real-time predictions:

- Input: email text
- Output: text (prediction result)
- Explanation: Users will input an email and the app will return the prediction result.

Additionally, I will be implementing model retraining if time allows:

- Option 1:
 - Input: CSV file (containing two columns - email text and spam/not spam label)
 - Output: text (confirmation of model retraining or error message)

- Explanation: Users will upload additional training data in the form of a CSV and will receive either confirmation of the model retraining or an error message.
- Additional Steps Needed:
 - I will first need to create an *additional Lambda function* that will handle the model retraining.
 - If the model retraining results in a model that is considerably less accurate than the original model (say by more than 2%), I will stick with the original model.
 - Then, I will need to implement a workflow for the different Lambda functions (simple prediction vs model retraining) by creating an *AWS Step Functions* state machine.
- Option 2:
 - Input: two text inputs (email text and spam/not spam label)
 - Output: text (prediction result + confirmation/error message that the new data has been stored)
 - Explanation: Users can input an email and the correct label, and the app will return the prediction result along with either a confirmation or an error message notifying them that their data has been stored and will be used to retrain the model in the future. Once enough new data has been collected, the model will be retrained.
 - Additional Steps Needed:
 - First, I will need to create an *additional Lambda function* that will either 1) append the new data to an existing retraining CSV file in S3, or 2) create the retraining CSV file with the new data if that file does not yet exist.
 - Then, I will need to create *another Lambda function* that will be triggered once the retraining CSV file has enough data. It will handle the model retraining and delete the retraining CSV file from S3 once retraining is completed.
 - If the model retraining results in a model that is considerably less accurate than the original model (say by more than 2%), I will stick with the original model.
 - Finally, I will need to implement a workflow for the different Lambda functions (simple prediction vs retraining data storage vs model retraining) by creating an *AWS Step Functions* state machine.

System monitoring:

I will be incorporating *AWS X-Ray*, which is a service that will collect data about requests that my application serves and provide tools I can use to view, filter, and gain insights into the data to identify issues and opportunities for optimization.

- Debugging: *AWS X-Ray* can give me an idea of what and where the issue is, but having logs is also important as they allow me to know what the request and response were and will contain a way to troubleshoot my application. Therefore, I will also be utilizing *Amazon CloudWatch Logs* to monitor, store, and access my log files from the many different services I will be using.

Explanation of Design Decisions:

Why I have chosen to design my system this way:

The main reason that I have chosen to design my system this way is due to its serverless architecture. Serverless architectures are scalable, highly available, and fully managed, all at a reduced cost.

Another reason I chose this design is that Amazon SageMaker allows you to upload a pretrained model. So if I have extra time, I can go back to the prototyping stage to further fine tune my model and simply upload the revised version of it to AWS instead of starting the whole process from scratch.

Why I have chosen to use certain technologies:

I am already familiar with AWS, having taken the “AWS Cloud Technical Essentials” and “Building Modern Python Applications on AWS” courses on Coursera. In the second of the aforementioned courses, I gained hands-on experience in implementing a serverless API-driven application on AWS. This API-driven application used Amazon API Gateway for serverless API hosting, Amazon S3 for data storage, AWS Lambda (paired with AWS SDKs for data processing and AWS Step Functions for an asynchronous workflow) for serverless computing, Amazon Cognito for serverless authentication, and AWS X-Ray for distributed tracing. My deployment design for this Capstone Project will utilize a lot of the same AWS services (API Gateway, S3, Lambda, AWS X-Ray, Amazon CloudWatch, and potentially AWS Step Functions).

Furthermore, I have recently gotten approval from AWS to rent their Amazon EC2 GPU instances, which are ideal for running CNNs on. However, I am unsure as to whether I will even need to use a GPU instance, as they are not included in AWS Free Tier and I want to keep costs low.

Estimated cost of the system:

I believe that the AWS Free Tier should suffice for my web application.


Deployment Steps:

Part 1: Deploy my trained CNN (Keras model) using Amazon SageMaker

1. Save and upload my Keras model in an S3 *bucket*:
 - a. To [save a Keras model as a zip file that includes JSON and weights](#):
 - i. `model.save('path/to/location.keras')`
 - b. Upload the zipped Keras model to an S3 bucket
2. To [deploy my trained model](#) using *SageMaker Serverless Endpoint*:
 - a. Create a notebook instance in the *SageMaker* console
 - i. Load the Keras model from the S3 bucket and export it to the TensorFlow ProtoBuf format

- ii. Convert TensorFlow model to an Amazon SageMaker-readable format and save in S3
 - iii. Instantiate the SageMaker TensorFlow serving model:
 1. `from sagemaker.tensorflow.serving import Model`
 # Instantiate the SageMaker TensorFlow serving model
 `model = Model(model_data=model_data,`
 `framework_version=tf_framework_version, role=role)`
 - iv. Deploy the model to Amazon Sagemaker Serverless endpoint by [calling the deploy method on the model and passing it a ServerlessInferenceConfig](#) object:
 1. `from sagemaker.serverless import ServerlessInferenceConfig`
 # Create an empty ServerlessInferenceConfig object to use default values
 `serverless_config = ServerlessInferenceConfig()`
 2. `text_classifier =`
 `model.deploy(serverless_inference_config=serverless_config)`
3. Copy down the name for our deployed SageMaker endpoint:
`print(text_classifier.endpoint)`

Resources:

- [Save, serialize, and export models | TensorFlow Core](#)
- [Deploy trained Keras or TensorFlow models using Amazon SageMaker | AWS Machine Learning Blog](#)
-  [Deploy your own pre-trained Keras model to aws Sagemaker](#)
- https://github.com/austinlasseter/blazing-text-dbpedia/blob/master/blazing_text_lab2.ipynb
- [Deploying to TensorFlow Serving Endpoints — sagemaker 2.175.0 documentation](#)

Part 2: Connect the SageMaker Endpoint to a serverless function using AWS Lambda and API Gateway

1. Create a *Lambda* function to do some preprocessing to transform HTML input data into a CSV file that the endpoint is expecting, and return the inference output in a format expected by API Gateway
 - a. Make sure to attach an IAM role that provides full access to our SageMaker endpoint
 - b. [Sample code](#) for the Lambda function
2. Use *API Gateway* to [build an HTTP endpoint that will allow our back-end Lambda function to talk to our front-end web app](#)
 - a. Make sure to copy the URL provided by API Gateway to invoke our endpoint
3. [Host the app](#) on an *S3 bucket*
 - a. [Sample HTML code](#) for web page

Resources:

- [Deploy an NLP classification model with Amazon SageMaker and Lambda | by Austin Lasseter](#)
- [Call an Amazon SageMaker model endpoint using Amazon API Gateway and AWS Lambda | AWS Machine Learning Blog](#)
- [Build an API Gateway REST API with Lambda integration](#)
- [How to host a static website on an Amazon S3 bucket | by Austin Lasseter](#)

Part 3: **Implement Model Retraining (if time allows)**

1. Decide between Option 1 and Option 2 (outlined under Inputs and Outputs in the General Overview section)

Resources:

- [Automate Model Retraining & Deployment Using the AWS Step Functions Data Science SDK](#)

Pre-deployment checklist:

1. Problem definition:
 - a. What is the problem?
 - i. Spam is unwanted and unsolicited messages sent electronically. My capstone project aims to create an ML model that is capable of detecting spam email messages.
 - b. Why does the problem need to be solved?
 - i. Spam messages often have malicious intent, and range from misleading advertising to phishing and malware spreads. Thus, spam is detrimental to both users and services, and creates mistrust and wariness between the two parties.
 - ii. Furthermore, spam is rapidly on the rise, with the Federal Trade Commission reporting \$8.8 billion in total reported losses in 2022, compared to the \$6.1 billion in 2021 and the mere \$1.2 billion in 2020 ([FTC.gov](#)). Therefore, it is increasingly important for companies and services to detect and filter spam messages.
 - c. How could the problem be solved manually?
 - i. Someone could read through each individual email to classify it as spam or not, but given that an average office employee sends approximately 40 emails daily and receives around 121 emails daily ([The Small Business Blog](#)) this will be no easy feat.

2. Data Preparation:

a. Data Description

- i. My dataset includes 39,763 entries, with 20,695 labeled as ham and 19,068 as spam and is made up of two premade datasets: [SpamAssassin dataset](#) and [Enron Spam dataset](#).

b. Data Wrangling:

- i. My text preprocessing/normalization process was as follows:
 1. Transform each token to lower case
 2. Replace URLs with the string 'URL'
 3. Replace emails with the string 'email'
 4. Replace numbers with the string 'number'
 5. Remove any extra newlines or whitespace
 6. Remove stopwords
 7. Remove non-ASCII characters
- ii. I chose not to remove punctuation as I believed it to be important in the detection of Spam emails.

c. Data Exploration:

- i. The key takeaways were as follows:
 1. My dataset was fairly balanced: 52% ham to 48% spam.
 2. The most common tokens were numbers, punctuation, and emails as well as the words 'enron', 'ect', 'company', and 'subject'.
 - a. Ham emails commonly featured emails and the words 'enron', 'subject', 'ect', and 'energy'.
 - b. Spam emails commonly featured urls and the words 'company', 'information', and 'font'.
 3. The average email length was around 201 tokens, but the longest email contained 28,624 tokens.

3. Spot Check Algorithms

- a. Create Test Harness: Completed
- b. Evaluate Candidate Algorithms: Completed

4. Improve Results

- a. Algorithm Tuning: Completed
- b. Ensemble Methods: Completed
- c. Model Selection: Completed

5. Finalize Project

- a. Present Results
- b. Operationalize Results