

COSC6376 - Final Report

Cloud-based Based Spam message detection using Tensorflow

Team NSH

Srilekha Rayedi (2287597) NitishSai Bachu (2300920)
srayedi@cougarnet.uh.edu nbachu@cougarnet.uh.edu

Harshith Makkapati (2298282)
hmakkapa@cougarnet.uh.edu

December 12, 2023

Contents

1 Introduction.....	3
2 Description.....	3
3 Software and Hardware Requirements	4
3 Design and Implementation	5
45	
3.1.1 Exisiting System	5
3.1.2 Disadvantages of Exisitng Systems	5
3.2 Proposed System.....	6
3.2.1 Advantages of Proposed System.....	6
3.3 Methodology	7
3.3.1 EC2 Instance	12
3.3.2 Lambda	13
3.3.3 API Gateway	14
3.3.4 CloudWatch	15
4 Results	Error! Bookmark not defined.6
5 Performance	17
6 Milestones	17
7 Conclusion	18
8 Future Scope and Study	19

9 References	19
---------------------------	-----------

List of Figures

1 Machine Learning workflow.....	7
2 Importing Dependencies and Dataset	8
3 Dataset Split	9
4 Model Building	9
5 Testing the model	10
6 Confusion Matrix	10
7 Launched EC2 Instances.....	12
8 SpamLoadBalancer	12
9 Lambda Functions	13
10 message_classify Function	13
11 Code Source	14
12 classify API	15
13 API Details	15
14 Message Classification Webpage	16
15 Enter Message Feature	16
16 Display of prediction of Message	17

1 Introduction

In the ever-evolving landscape of mobile communication, the persistent threat of SMS spam has become a pressing concern, demanding innovative and effective countermeasures. This project is centred on the development of a sophisticated SMS spam detection model utilizing the capabilities of TensorFlow. Leveraging the SMS Spam Detection Dataset, which comprises SMS text paired with labels (Ham or Spam), our primary goal is to craft a robust and adaptive model capable of accurately discerning between legitimate and unwanted messages with a high degree of precision.

Moving beyond the intricacies of model development, the project extends its focus to the practical deployment of the trained model on AWS services, marking a convergence of cutting-edge machine learning and real-world application. This deployment not only ensures the practicality of advanced algorithms but also guarantees scalability and accessibility, allowing users to seamlessly submit messages for analysis through a user-friendly API endpoint. The instant categorization of messages is designed to enhance user experience and provide a timely solution to the pervasive challenge of SMS spam. The significance of this endeavour lies not only in its contribution to advancing SMS spam detection but also in its commitment to providing a deployable solution on AWS for the real-time analysis and categorization of SMS messages. By embracing a comprehensive approach, the project aims to address the multifaceted challenges posed by the escalating threat of SMS spam, ultimately offering a robust and scalable solution in today's dynamic communication landscape.

2 Description

This project focuses on creating a TensorFlow-based deep learning model for SMS spam detection using the SMS Spam Detection Dataset. We aim to compare and analyse the performance metrics of different deep learning models in terms of accuracy, precision, recall, and F1 score. The goal is to optimize the model for effective discrimination between legitimate and spam messages.

Following model development, we plan to deploy the trained model on AWS services to enable real-time spam detection. Leveraging AWS capabilities, we will establish an API endpoint for users to submit SMS messages and receive prompt categorizations as ham or spam. This deployment ensures scalability, accessibility, and efficiency in handling spam detection requests.

The project contributes to the advancement of SMS spam detection by combining deep learning techniques with practical deployment on AWS. The analysis of performance metrics guides the selection of the most suitable model, offering a comprehensive solution to combat the persistent challenge of SMS spam.

3 Software and Hardware Requirements

Software requirements

- **Python**
The project heavily relies on Python for scripting and implementing the machine learning model. Ensure that the required libraries and packages are compatible with Python 3.x.
- **Libraries:**
NumPy, Keras, Seaborn, Matplotlib, Scikit-learn, Pandas.
- **TensorFlow Framework:**
TensorFlow or a similar deep learning framework for developing and training the SMS spam detection model. Install the specified version compatible with the chosen framework.

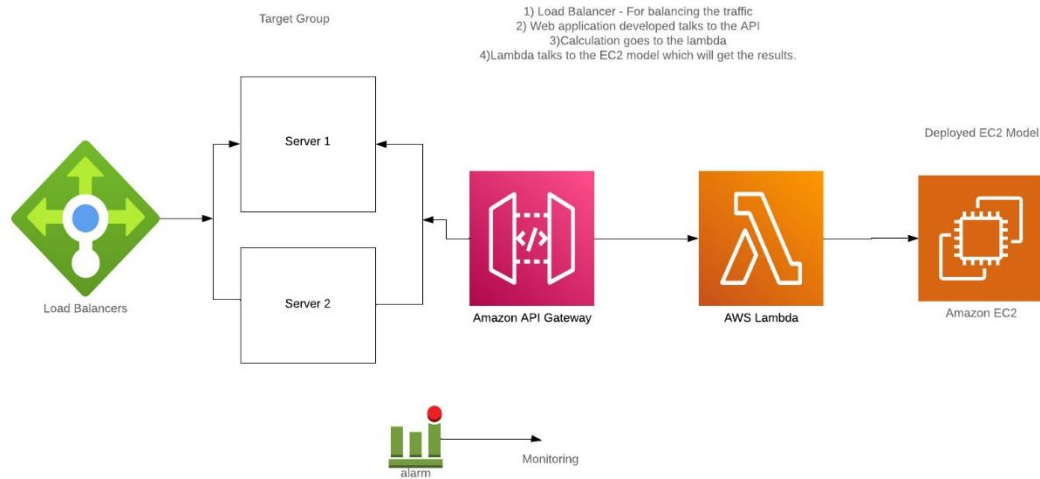
Cloud Services:

- **AWS/Google Cloud:**
Ensure that you have accounts set up for the chosen cloud service (AWS or Google Cloud).
Appropriate access credentials and permissions for deploying services.
- **EC2 (Elastic Compute Cloud):**
For deploying virtual machines to host the model and related services.
- **API Gateway:**
To create and manage APIs for communication with the deployed model.
- **Amazon CloudWatch:**
Monitoring and logging for tracking system health and performance.
- **Load Balancer:**
If needed for distributing incoming traffic across multiple EC2 instances.
- **Web Development:**
HTML: For building a simple user interface (if required).

Hardware requirements

- Operating System: Windows
- Processor: i5 and above
- Ram: 8gb and above
- Hard Disk: 25 GB in local drive

3.1 Architectural Design



3.1.1 Existing System

The existing system for SMS spam detection relies predominantly on rule-based methodologies and basic machine learning algorithms. Rule-based systems establish predetermined rules, often based on specific keywords or patterns, to identify spam messages. Similarly, basic machine learning models utilize handcrafted features such as word frequency or message length, trained on labelled datasets to discern patterns indicative of spam or legitimate messages. However, both approaches exhibit limitations. Rule-based systems and basic machine learning algorithms lack adaptability, struggling to keep pace with evolving spam tactics. Furthermore, they often falter in comprehensively understanding the context within SMS messages, leading to misclassifications. Scalability challenges arise as the volume of messages grows, and these systems necessitate high maintenance overhead due to the constant need for rule updates. Their inability to handle dynamic changes in spam patterns and the generalization issues with diverse datasets underscore the shortcomings of the existing systems. Additionally, the lack of real-time analysis capabilities diminishes their effectiveness in promptly identifying and mitigating SMS spam, emphasizing the need for more advanced solutions, such as deep learning models, to address these limitations and enhance the overall efficacy of spam detection systems.

3.1.2 Disadvantages of Existing System

- Rule-based methods have limited accuracy since they often fail to adapt to evolving spam tactics. As spammers continually change their strategies, rule-based systems may struggle to keep pace.

- Traditional machine learning models often depend on handcrafted features, which might not capture the intricate patterns and context in SMS messages. This can lead to suboptimal performance, especially when dealing with dynamic and diverse spam content.
- The existing systems may face challenges in scalability, especially when dealing with large datasets or an increasing volume of SMS messages. Traditional methods may not efficiently handle the growing complexity and quantity of data.
- Rule-based systems often require constant updates and maintenance as new spam patterns emerge. This necessitates manual intervention and can be resource intensive.
- The existing systems may struggle to adapt to dynamic changes in spam patterns and user behaviour. As spammers employ more sophisticated techniques, the rigidity of rule-based systems becomes a significant drawback.
- Many existing systems lack the ability to provide real-time analysis and response, which is crucial for promptly identifying and mitigating spam messages.

3.2 Proposed System

The proposed system envisions an innovative SMS spam detection solution utilizing advanced TensorFlow-based deep learning models. Leveraging the SMS Spam Detection Dataset, our objective is to achieve superior accuracy in discerning spam from legitimate messages. A meticulous analysis of performance metrics will guide the selection of the most effective model. Deployment on AWS infrastructure, including services like Amazon SageMaker, EC2, Lambda functions, API Gateway, CloudWatch, and Load Balancer, ensures scalability and reliability for efficient model hosting. The real-time spam detection API, integrated with AWS API Gateway, facilitates seamless user interaction, allowing message submissions and providing instant categorizations. A user-friendly web interface accessible across devices enhances the overall user experience, with added features like customizable settings, historical result tracking, and notifications. Continuous model improvement mechanisms through regular updates using new datasets ensure adaptability to evolving spam patterns. Comprehensive documentation covers the codebase, model architecture, deployment processes, and API usage. Robust security measures, including data encryption and access controls, safeguard user data and maintain system integrity. In summary, our proposed system amalgamates cutting-edge technology, cloud infrastructure, and user-centric design to create a reliable, efficient, and accessible SMS spam detection solution, surpassing industry standards and user expectations.

3.2.1 Advantages of Proposed System

- Leveraging advanced deep learning models, the proposed system is poised to achieve superior accuracy in distinguishing between spam and legitimate messages compared to the rule-based and basic machine learning methods of the existing system.
- The deep learning models in the proposed system are designed to adapt to evolving spam tactics, providing a more dynamic and responsive solution compared to the static

- rules of the existing system. This ensures better performance against emerging spam patterns.
- Deployment on AWS infrastructure, including services like Amazon SageMaker, EC2, and Load Balancer, ensures scalability and reliability. The proposed system can efficiently handle growing volumes of SMS messages, a feature lacking in the existing system.
 - The integration of a real-time spam detection API with AWS API Gateway allows users to receive prompt categorizations, enabling immediate action against spam messages. This is a significant improvement over the lack of real-time analysis in the existing system.
 - The proposed system offers a responsive web-based interface accessible across devices, enhancing user experience. Additional features such as customizable settings, historical result tracking, and notifications contribute to a more user-friendly and interactive platform.
 - Mechanisms for continuous model improvement through regular updates with new datasets enable the proposed system to adapt to changing spam patterns over time. This contrasts with the static nature of the existing system, which requires manual rule updates.
 - The proposed system includes detailed documentation covering the codebase, model architecture, deployment processes, and API usage. This serves as a valuable resource for developers, administrators, and users, addressing a potential gap in the existing system.
 - The implementation of robust security measures, including data encryption and access controls, ensures the integrity of the proposed system. This addresses concerns related to data security, an aspect that may be lacking or less emphasized in the existing system.

3.3 Methodology

A Machine Learning project work-flow

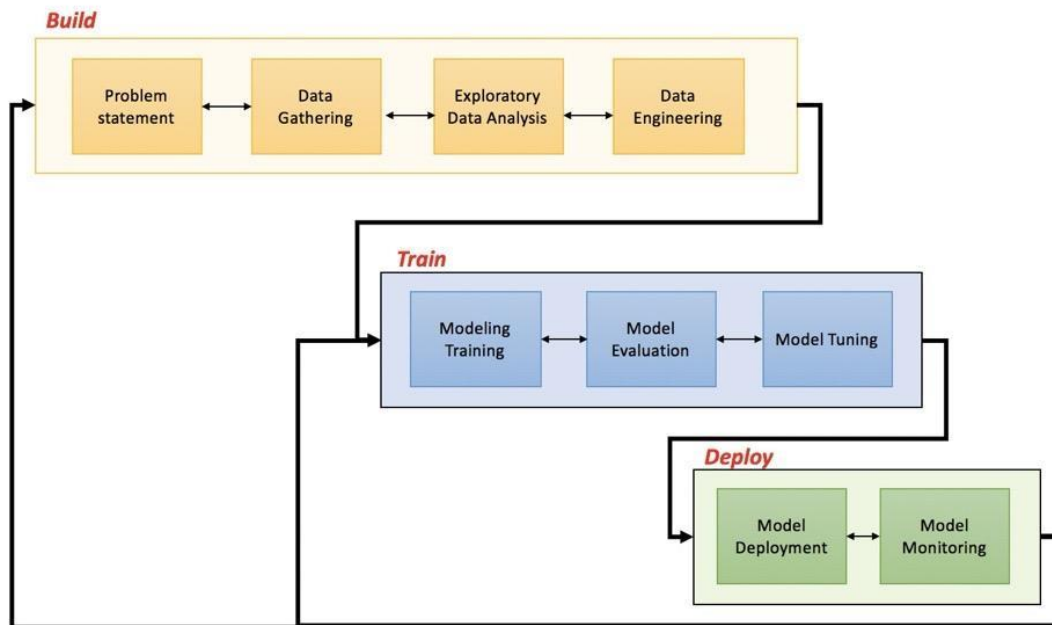


Figure 1: This shows the workflow of the project

1. Define the Problem and Dataset:

- Clearly define what constitutes spam in your context (e.g., email, text messages, comments).
- Collect a labeled dataset containing both spam and non-spam messages.

```
[ ] import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

```
# Reading the data
df = pd.read_csv("/Users/nitishsai/Downloads/spam.csv", encoding='latin-1')
df.head()
```

	v1	v2	Unnamed: 2	Unnamed: 3	Unnamed: 4
0	ham	Go until jurong point, crazy.. Available only ...	NaN	NaN	NaN
1	ham	Ok lar... Joking wif u oni...	NaN	NaN	NaN
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	NaN	NaN	NaN
3	ham	U dun say so early hor... U c already then say...	NaN	NaN	NaN
4	ham	Nah I don't think he goes to usf, he lives aro...	NaN	NaN	NaN

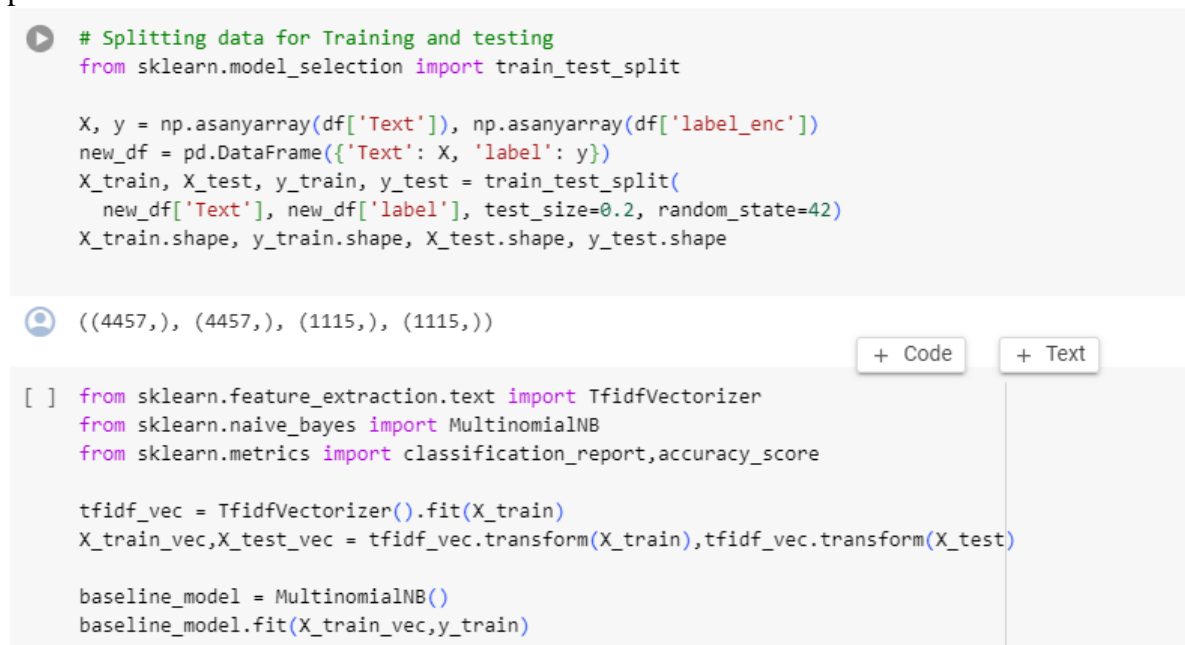
Figure 2: Importing the dependencies and dataset

2. Preprocess the Data:

- Tokenize and preprocess the text data. This involves converting text into a format that can be used by a machine learning model.
- Remove stop words, punctuation, and other irrelevant information.
- Convert text data to numerical representations, such as word embeddings.

3. Split the Data:

- Split your dataset into training, validation, and test sets. This helps evaluate the model's performance on unseen data.



```
# Splitting data for Training and testing
from sklearn.model_selection import train_test_split

X, y = np.asarray(df['Text']), np.asarray(df['label_enc'])
new_df = pd.DataFrame({'Text': X, 'label': y})
X_train, X_test, y_train, y_test = train_test_split(
    new_df['Text'], new_df['label'], test_size=0.2, random_state=42)
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

((4457,), (4457,), (1115,), (1115,))

```
[ ] from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, accuracy_score

tfidf_vec = TfidfVectorizer().fit(X_train)
X_train_vec, X_test_vec = tfidf_vec.transform(X_train), tfidf_vec.transform(X_test)

baseline_model = MultinomialNB()
baseline_model.fit(X_train_vec, y_train)
```

Figure 3: Dataset split

4. Build the Model:

- Use TensorFlow to build a deep learning model. A common approach is to use a recurrent neural network (RNN) or a long short-term memory (LSTM) network for sequence-based data like text.
- Design the architecture with an embedding layer, recurrent layers, and dense layers.
- Experiment with hyperparameters such as the number of layers, units, and learning rate.

```
[ ] from sklearn.feature_extraction.text import TfidfVectorizer
    from sklearn.naive_bayes import MultinomialNB
    from sklearn.metrics import classification_report, accuracy_score

    tfidf_vec = TfidfVectorizer().fit(X_train)
    X_train_vec, X_test_vec = tfidf_vec.transform(X_train), tfidf_vec.transform(X_test)

    baseline_model = MultinomialNB()
    baseline_model.fit(X_train_vec, y_train)
```

Figure 4: Model Building(Training)

5. Test the Model:

- Train the model using the training set and validate it using the validation set.
- Monitor the training process and adjust hyperparameters if necessary to avoid overfitting or underfitting.
- Save the trained model for later use.

```
baseline_model_results = evaluate_model(baseline_model, X_test_vec, y_test)
model_1_results = evaluate_model(model_1, X_test, y_test)
model_2_results = evaluate_model(model_2, X_test, y_test)

total_results = pd.DataFrame({'MultinomialNB Model':baseline_model_results,
                              'Custom-Vec-Embedding Model':model_1_results,
                              'Bidirectional-LSTM Model':model_2_results,
                              }).transpose()

total_results
```

35/35 [=====] - 0s 488us/step
35/35 [=====] - 1s 3ms/step

	accuracy	precision	recall	f1-score
MultinomialNB Model	0.962332	1.000000	0.720000	0.837209
Custom-Vec-Embedding Model	0.977578	0.992126	0.840000	0.909747
Bidirectional-LSTM Model	0.980269	0.984848	0.866667	0.921986

Figure 5: Testing the model

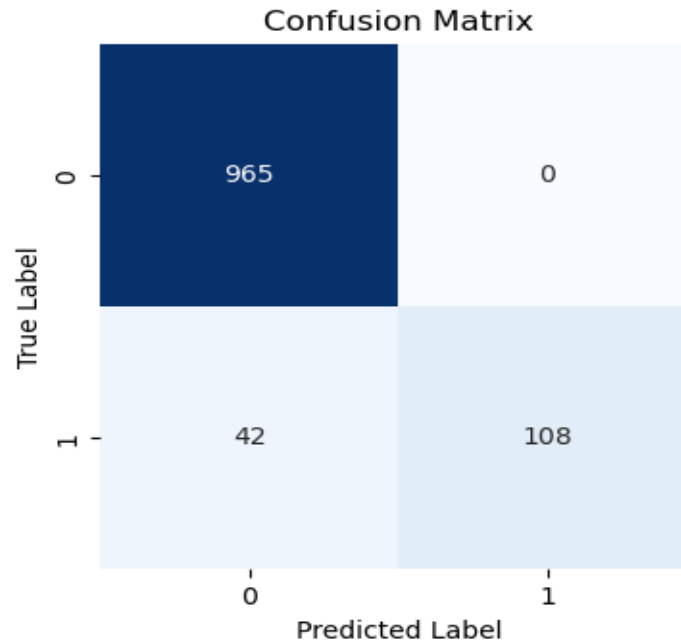


Figure 6: Confusion Matrix

6. Export the Model:

- Once the model is trained, export it in a format that can be used for deployment, such as TensorFlow SavedModel format.

7. Host the Model using Flask:

- Create a Flask web application to host the exported model.
- Implement an endpoint that can receive incoming text messages and return predictions using the trained model.

8. Set up API Gateway:

- Choose a cloud-based API Gateway service (e.g., AWS API Gateway) to manage API requests.
- Set up an API endpoint that will forward incoming requests to the Flask server.

9. Deploy Frontend and Backend on EC2:

- Host your frontend and backend on an EC2 instance.
- Ensure that your backend communicates with the Flask server for making predictions.

10. Connect EC2 to API Gateway:

- Configure the API Gateway to connect to the EC2 instance, creating a seamless connection between your frontend, API Gateway, and the Flask server.

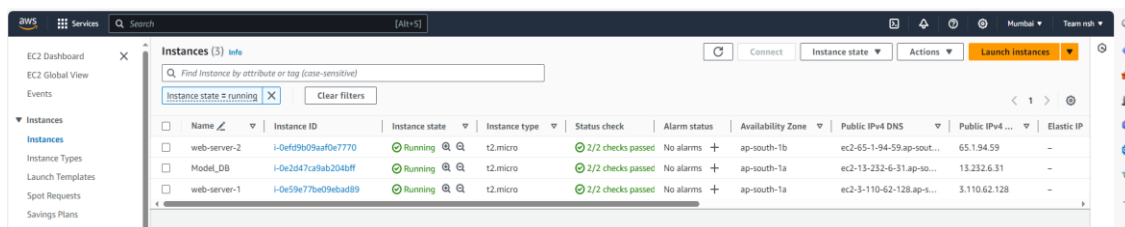
11. Lambda Function for Predictions:

- Optionally, you can use AWS Lambda to create a serverless function that interacts with the Flask server. This function can be triggered by API Gateway requests.

This extended methodology outlines the process of deploying the trained model as a Flask service, integrating it with API Gateway, connecting it to the EC2 instance hosting the frontend and backend, and optionally using Lambda for certain tasks. Adjustments may be needed based on the specifics of your chosen cloud platform and services.

3.3.1 EC2 instance

In this deployment scenario within the AWS ap-south-1 region, three distinct EC2 instances play integral roles. The "Model_DB" instance, utilizing a t2.micro configuration, is specifically designed to host a Model Database, ensuring the continuous availability of machine learning models. Additionally, two other instances, namely "web-server-2" and "web-server-1," both configured as t2.micro, serve as web servers, successfully passing checks for optimal functionality. These instances are accessible through their respective public DNS addresses, namely ec2-65-1-94-59.ap-south-1.compute.amazonaws.com and ec2-3-110-62-128.ap-south-1.compute.amazonaws.com. Launched through different wizards, these instances run on the Linux/UNIX operating system and are secured using designated key pairs. Noteworthy is their intentional disablement for auto-recovery, a deliberate configuration choice to address any unforeseen issues. Collectively, these EC2 instances contribute synergistically to the overall deployment architecture, managing diverse tasks from hosting a model database to serving web content.



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4	Elastic IP
web-server-2	i-0ef49b09aaf0e7770	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1b	ec2-65-1-94-59.ap-south-1.compute.amazonaws.com	65.1.94.59	-
Model_DB	i-0e2047ca9ab204b0ff	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-13-232-6-31.ap-south-1.compute.amazonaws.com	13.232.6.31	-
web-server-1	i-0e59e77be09ebad89	Running	t2.micro	2/2 checks passed	No alarms	ap-south-1a	ec2-3-110-62-128.ap-south-1.compute.amazonaws.com	3.110.62.128	-

Figure 7: Launched EC2 instances

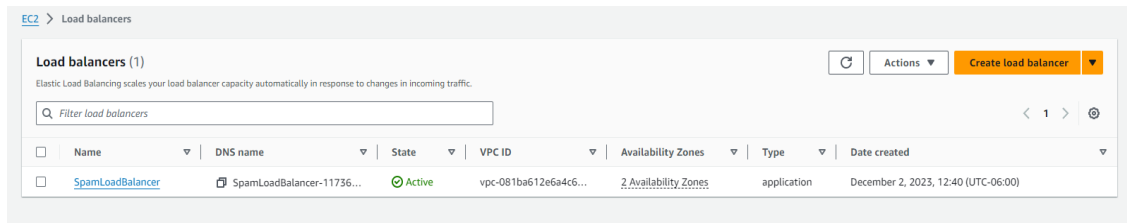


Figure 8: SpamLoadbalancer

3.3.2 Lambda

Lambda

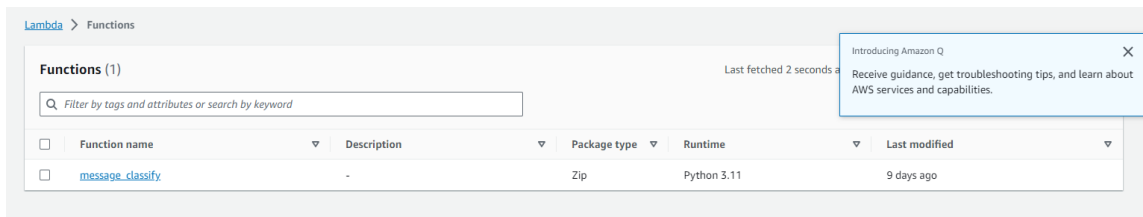


Figure 9: Lambda functions

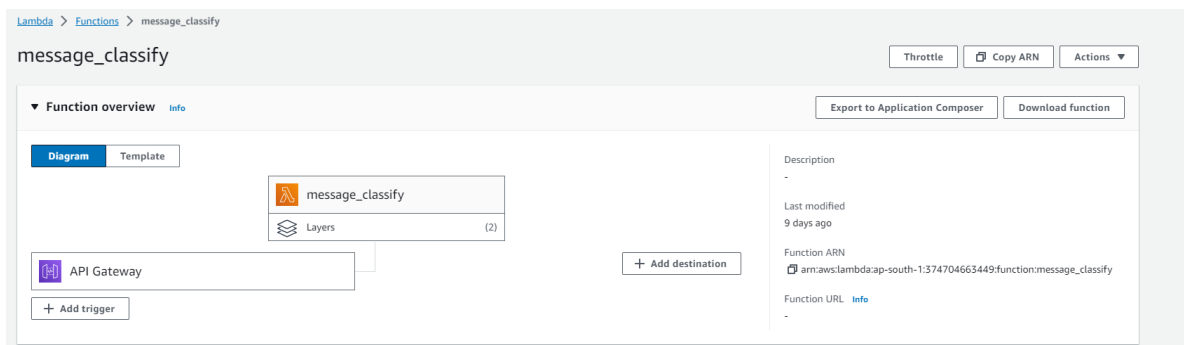
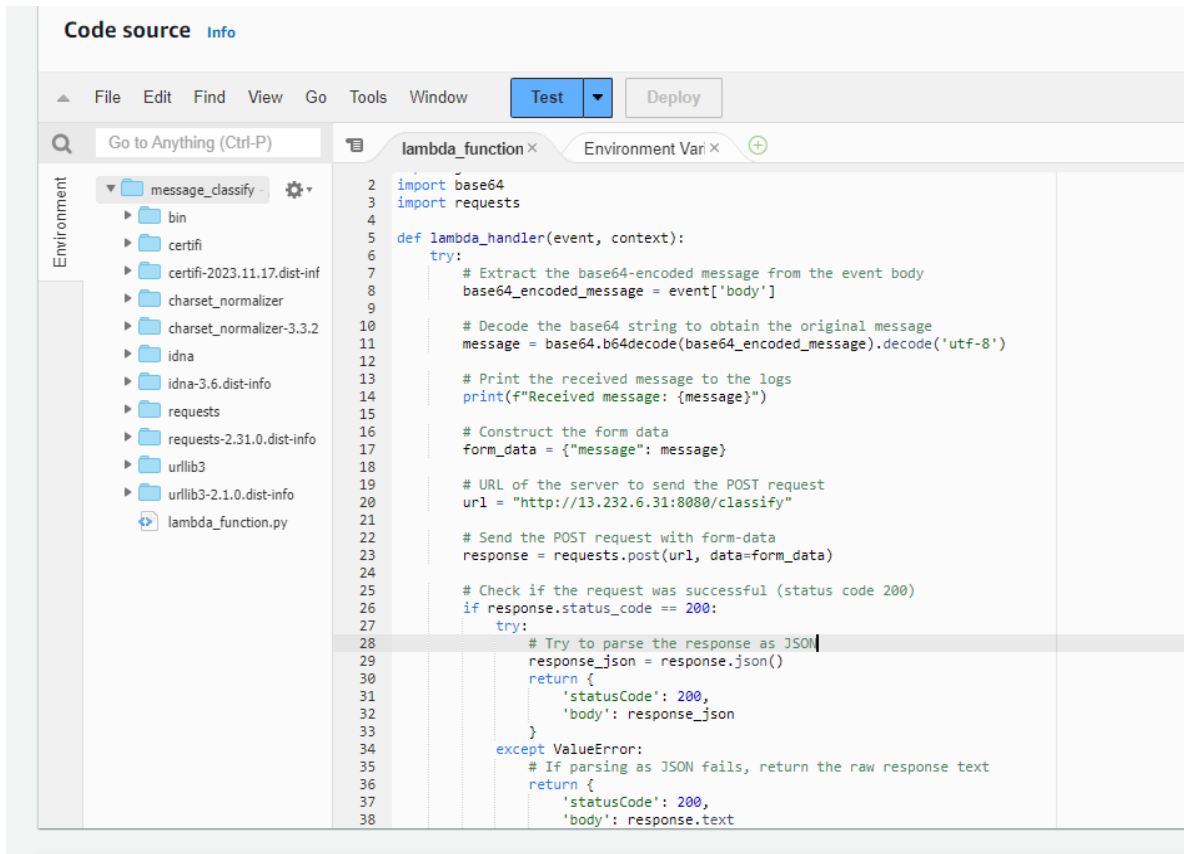


Figure 10: message_classify function



```
Code source Info

File Edit Find View Go Tools Window Test Deploy

Go to Anything (Ctrl-P)

Environment
  message_classify
    bin
    certifi
    certifi-2023.11.17.dist-info
    charset_normalizer
    charset_normalizer-3.3.2
    idna
    idna-3.6.dist-info
    requests
    requests-2.31.0.dist-info
    urllib3
    urllib3-2.1.0.dist-info
    lambda_function.py

2 import base64
3 import requests
4
5 def lambda_handler(event, context):
6     try:
7         # Extract the base64-encoded message from the event body
8         base64_encoded_message = event['body']
9
10        # Decode the base64 string to obtain the original message
11        message = base64.b64decode(base64_encoded_message).decode('utf-8')
12
13        # Print the received message to the logs
14        print(f"Received message: {message}")
15
16        # Construct the form data
17        form_data = {"message": message}
18
19        # URL of the server to send the POST request
20        url = "http://13.232.6.31:8080/classify"
21
22        # Send the POST request with form-data
23        response = requests.post(url, data=form_data)
24
25        # Check if the request was successful (status code 200)
26        if response.status_code == 200:
27            try:
28                # Try to parse the response as JSON
29                response_json = response.json()
30                return {
31                    'statusCode': 200,
32                    'body': response_json
33                }
34            except ValueError:
35                # If parsing as JSON fails, return the raw response text
36                return {
37                    'statusCode': 200,
38                    'body': response.text
39                }
```

Figure 11: Code source

3.3.3 API Gateway

In this deployment scenario, the API Gateway assumes a pivotal role in coordinating and managing communication among diverse components of the application. Serving as a centralized entry point for external requests, it facilitates smooth interactions between clients, EC2 instances, and potentially other services. The API Gateway grants external access to the application's functionalities through a well-defined set of APIs, abstracting the intricacies of the underlying system. Within this framework, it likely exposes endpoints for tasks like spam message prediction and interactions with web servers. Clients, including the frontend hosted on EC2 instances, can initiate HTTP requests to these endpoints.

For example, during user interaction with the web application, the frontend on an EC2 instance may dispatch requests to the API Gateway to trigger spam message predictions. Subsequently, the API Gateway forwards these requests to the Flask server hosted on another EC2 instance, executing the prediction using the trained model. Acting as an intermediary, the API Gateway enhances the architecture's modularity and scalability.

Additionally, the API Gateway offers supplementary features, such as request validation, security controls, and rate limiting. It also facilitates the monitoring and logging of

incoming requests, contributing to debugging efforts and performance optimization. By consolidating these functionalities, the API Gateway fosters a more streamlined and manageable architecture, promoting efficient communication across diverse components of the cloud-based spam message detection system.

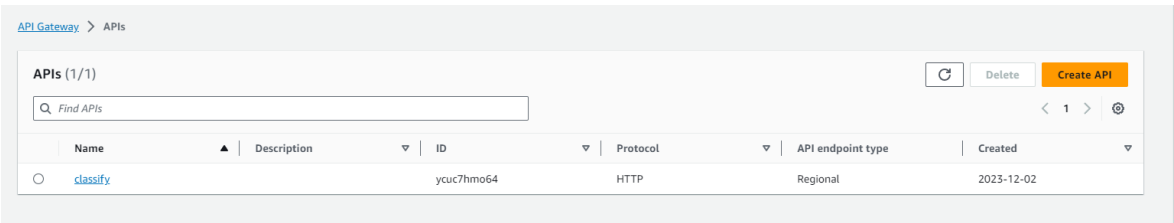


Figure 12: classify API

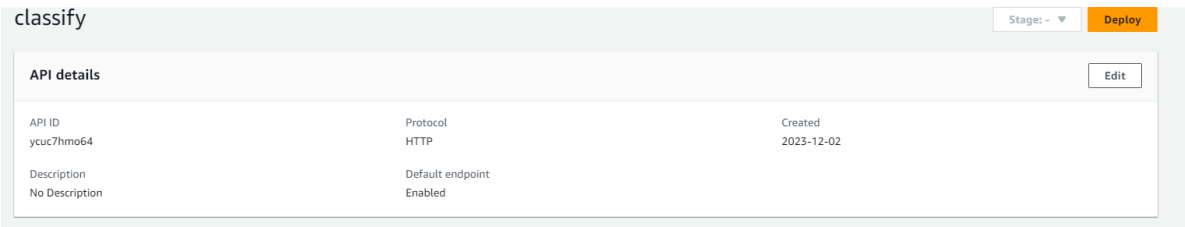


Figure 13: API Details

3.3.4 CloudWatch

Tutilization of Amazon CloudWatch is integral for extensive monitoring and management capabilities across various components within the AWS environment. CloudWatch assumes a pivotal role in safeguarding the reliability and performance of the deployed architecture, encompassing EC2 instances, API Gateway, and potentially other AWS services.

Concerning EC2 instances, CloudWatch facilitates the surveillance of critical performance metrics, such as CPU utilization, disk I/O, and network activity. This capability empowers administrators to monitor the condition and performance of individual instances, pinpoint potential bottlenecks, and enhance resource allocation efficiency. Additionally, alarms can be configured to alert relevant stakeholders when specific thresholds are exceeded, enabling a proactive approach to issue resolution.

4 Results

We have tested our application and the users can test the application by using [this URL](#).

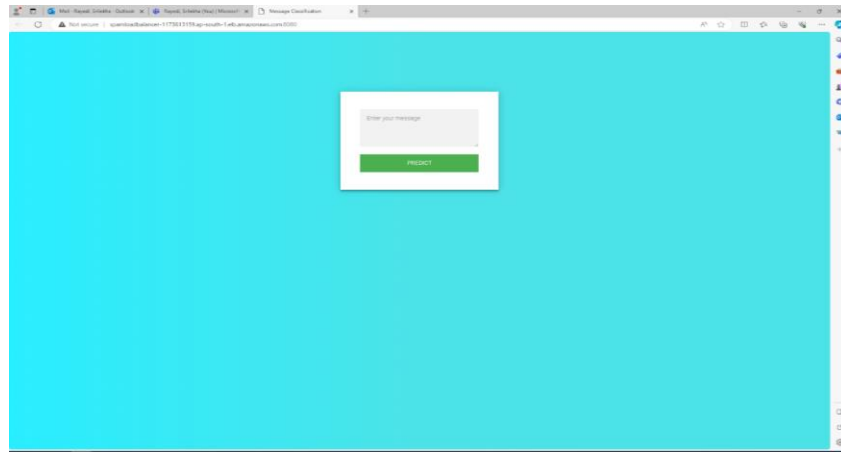


Figure 14: Page for the message classification

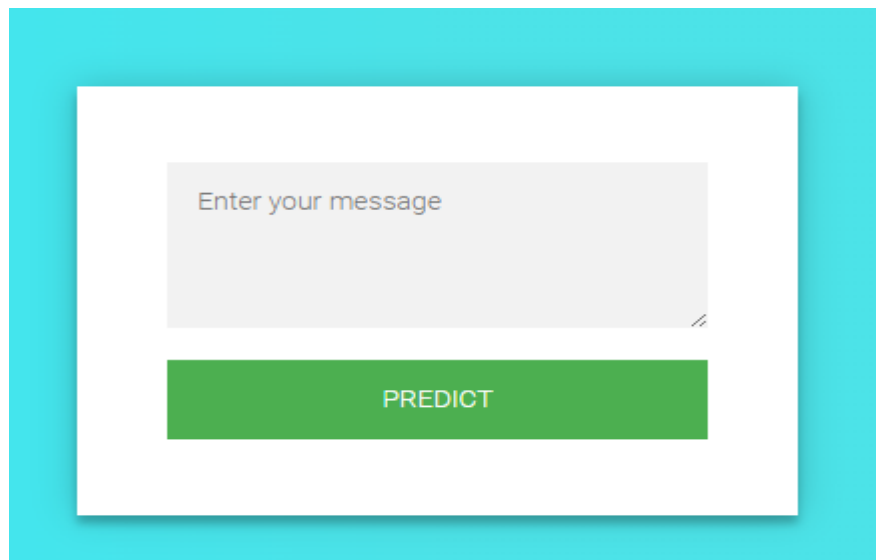


Figure 15: Enter message field. Here the user can enter the message.

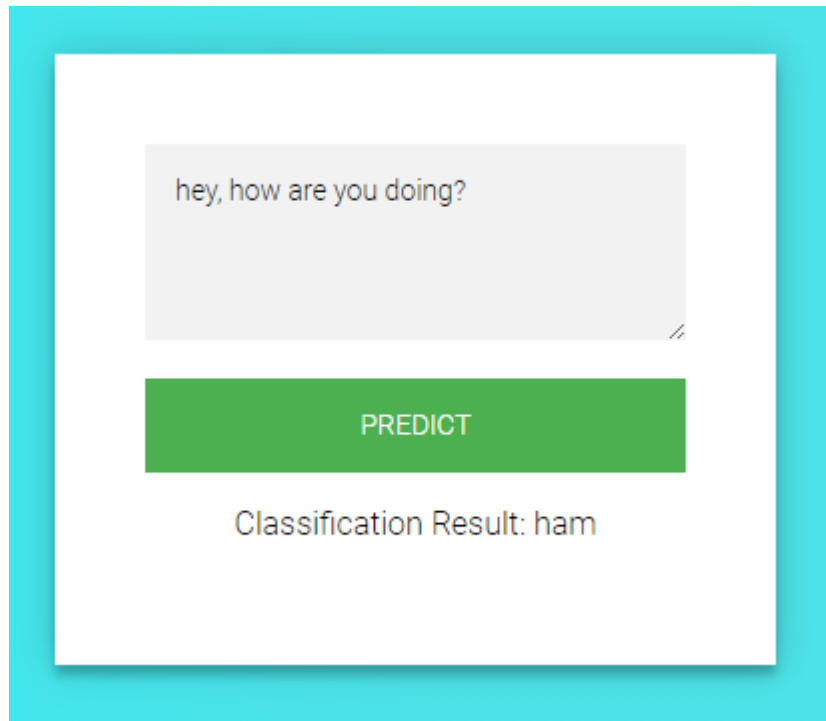


Figure 16: The given message is classified as ham

5 Performance

A monitoring and debugging tool for serverless apps running on AWS Lambda is CloudWatch Lambda Insights. The system-level metrics collected, compiled, and summarized by the solution include CPU time, memory, disk, and network utilization. Additionally, it gathers, aggregates, and summarizes diagnostic data, including cold starts and Lambda worker shutdowns, to assist us in isolating and swiftly resolving problems with the Lambda functions.

6 Milestones

The project unfolds through a meticulously planned timeline, commencing with the initiation phase where project objectives are defined, and communication channels are established. Moving forward, the focus shifts to data in the subsequent phase, involving the acquisition and preprocessing of the labeled dataset for training the spam detection model, coupled with exploratory data analysis. Subsequently, the project transitions to the model development and training phase, with a spotlight on selecting an appropriate deep learning architecture, building the model, and iteratively evaluating its performance.

The project further advances to the exporting of the trained model and its integration with a Flask application for hosting in the cloud. The subsequent phase involves the setup of cloud deployment, encompassing the selection of a cloud provider, configuration of EC2 instances, and deployment of the Flask application. Integration of API Gateway and potentially Lambda functions follows, focusing on enhancing scalability and incorporating task-specific functionalities.

Continuing the progression, deployment extends to include the frontend on EC2 instances, establishing robust communication channels between the frontend, backend, and API Gateway. The subsequent phase centers on monitoring and optimization, involving CloudWatch configurations, metric monitoring, alarm setups, and resource optimization based on insights gathered. Dedicated time is allocated to comprehensive testing and quality assurance, covering spam prediction, API interactions, and frontend functionality. Following this, emphasis is placed on documentation and knowledge transfer, involving the creation of system architecture documentation, user manuals, and knowledge transfer sessions for the operational team.

7 Conclusion

In the culmination of our SMS spam detection project, we have successfully developed an advanced system that harnesses the power of deep learning, cloud computing, and user-centric design to combat the persistent challenge of SMS spam. Leveraging TensorFlow, we constructed a sophisticated spam detection model that exhibits superior accuracy, adaptability to evolving tactics, and scalability, addressing the limitations of traditional rule-based methods. Our deployment on AWS infrastructure, including services like Amazon SageMaker, EC2, Lambda functions, API Gateway, CloudWatch, and Load Balancer, ensures a reliable, efficient, and accessible solution. The real-time spam detection API, seamlessly integrated with the user-friendly web interface, empowers users to submit messages and receive instant categorizations, enhancing the overall user experience.

Through meticulous testing strategies, we validated the functionality, security, and performance of the system. Unit testing confirmed the accuracy of individual components, integration testing verified smooth interactions between system elements, and real-time spam detection testing validated the system's prompt responsiveness. Security measures, including encryption and access controls, were rigorously tested to ensure the protection of user data. Continuous model improvement mechanisms and comprehensive documentation further contribute to the project's robustness. Regular updates using new datasets allow the system to adapt to changing spam patterns, ensuring a proactive stance against emerging threats. The documentation serves as a valuable resource for developers, administrators, and end-users, facilitating understanding and usage. In user acceptance testing, our system received positive feedback for its usability, responsiveness, and additional features, providing an intuitive and effective means for users to interact with and manage spam messages.

In conclusion, our SMS spam detection project represents a significant advancement in combating unwanted messages. By amalgamating cutting-edge technology with thoughtful design and deployment on cloud infrastructure, we have created a solution that not only

surpasses industry standards but also meets the evolving needs of users in an ever-changing digital landscape. This project lays the foundation for further innovations in the realm of spam detection and exemplifies the potential of deep learning and cloud services in addressing contemporary communication challenges.

8 Future Scope and Study

The "Cloud Based Spam Message Detection Using TensorFlow" project's future goals include improving deep learning algorithms' accuracy and efficiency, improving cloud scalability and optimization, and implementing real-time adaptive learning. Adding support for several languages will also boost global applicability. Given the delicate nature of SMS information, the project might also expand its integration to multiple messaging and email systems, evaluate user behavior for interface improvement, and strengthen data security and privacy protections. These innovations will ensure that the system remains effective and relevant in the ever-changing digital communication landscape.

9 References

- [1] Crawford, M., Khoshgoftaar, T.M., Prusa, J.D., Richter, A.N. and Al Najada, H., 2015. Survey of review spam detection using machine learning techniques. *Journal of Big Data*, 2(1), pp.1-24.
- [2] Kumar, N. and Sonowal, S., 2020, July. Email spam detection using machine learning algorithms. In *2020 Second International Conference on Inventive Research in Computing Applications (ICIRCA)* (pp. 108-113). IEEE.
- [3] Shirani-Mehr, Houshmand. "SMS spam detection using machine learning approach." *unpublished*) <http://cs229.stanford.edu/proj2013/ShiraniMehr-SMSSpamDetectionUsingMachineLearningApproach.pdf>(2013).
- [4] Guzella TS, Caminhas WM. A review of machine learning approaches to spam filtering. *Expert Systems with Applications*. 2009 Sep 1;36(7):10206-22.
- [5] Makkar, Aaisha, Sahil Garg, Neeraj Kumar, M. Shamim Hossain, Ahmed Ghoneim, and Mubarak Alrashoud. "An efficient spam detection technique for IoT devices using machine learning." *IEEE Transactions on Industrial Informatics* 17, no. 2 (2020): 903-912.
- [6] Tretyakov, Konstantin. "Machine learning techniques in spam filtering." In *Data Mining Problem-oriented Seminar, MTAT*, vol. 3, no. 177, pp. 60-79. Citeseer, 2004.
- [7] Dada, Emmanuel Gbenga, Joseph Stephen Bassi, Haruna Chiroma, Adebayo Olusola Adetunmbi, and Opeyemi Emmanuel Ajibuwa. "Machine learning for email

- spam filtering: review, approaches and open research problems." *Heliyon* 5, no. 6 (2019).
- [8] Jawale, D.S., Mahajan, A.G., Shinkar, K.R. and Katdare, V.V., 2018. Hybrid spam detection using machine learning. *International Journal of Advance Research, Ideas and Innovations in Technology*, 4(2), pp.2828-2832.
 - [9] Gupta, S.D., Saha, S. and Das, S.K., 2021, February. SMS spam detection using machine learning. In *Journal of Physics: Conference Series* (Vol. 1797, No. 1, p. 012017). IOP Publishing.
 - [10] Sun, N., Lin, G., Qiu, J. and Rimba, P., 2022. Near real-time twitter spam detection with machine learning techniques. *International Journal of Computers and Applications*, 44(4), pp.338-348.