CSP-554 Big Data Technologies

Final Project

Lipi Maheshbhai Shah A20455192
Nicholas Saveas A20374237
Manasdeep Deb A20449643
Yin Yang A20450316

Contents

Page #

I. **Abstract** 2

II. **Profile the Data** ..... 2 - 6
   a. Statistics of Features
   b. Correlation Matrix
   c. Boxplots

III. **Literature Review** ..... 6 - 8
   a. Sagemaker
   b. H2O
   c. SparkML
   d. General Architecture of Project

IV. **Project Execution/Results** ..... 8 - 28
   a. Sagemaker
   b. AWS Lambda
   c. AWS API Gateway
   d. AWS SNS
   e. Postman Testing for API URL
   f. SNS Topic SMS Service Output
   g. *Supplemental* - H2O

V. **References** ..... 28 - 29

I. Abstract

Breast cancer is a disease in which cells in the breast grow out of control. Among all unveiled cancers so far, breast cancer is the second most common cancer in women after skin cancer. The automatic disease detection system can help medical personnel diagnose diseases and provide reliable, effective, and rapid response, which reduces the risk of death.

In this project, we aim to focus on using AWS Sagemaker to build data pipelines and support end-to-end data science projects, and as a complement, we further try to perform machine learning analysis on H2O clusters. As a review of our literature, we compared SageMaker, SparkML and H2O.

II. Profile the Data

Breast Cancer Wisconsin (Diagnostic) Data Set - https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

This data set contains diagnostic information

- ID number
- Diagnosis - B = benign (total 357 cases)
  M = malignant (total 212 cases)

and **30 features** that describe the characteristics of the cell nucleus present in the digital image of fine-needle aspiration (FNA) of the breast.

There are ten real-valued features that are computed for each cell nucleus:

a. Radius - distances from the center to points on the perimeter.
b. Texture - standard deviation of gray-scale values.
c. Perimeter.
d. Area.
e. Smoothness - local variation in radius lengths.
f. Compactness - (perimeter / area - 1.0).
g. Concavity - the severity of concave portions of the contour.
h. Concave points - number of concave portions of the contour.
i. Symmetry.
j. Fractal dimension - ("coastline approximation" - 1).

The mean, standard error (SE) and worst(mean of the three largest values) of these features were computed for each image, which is 3 columns for each of the 10 values -- 30 features.

a. Statistics of Features

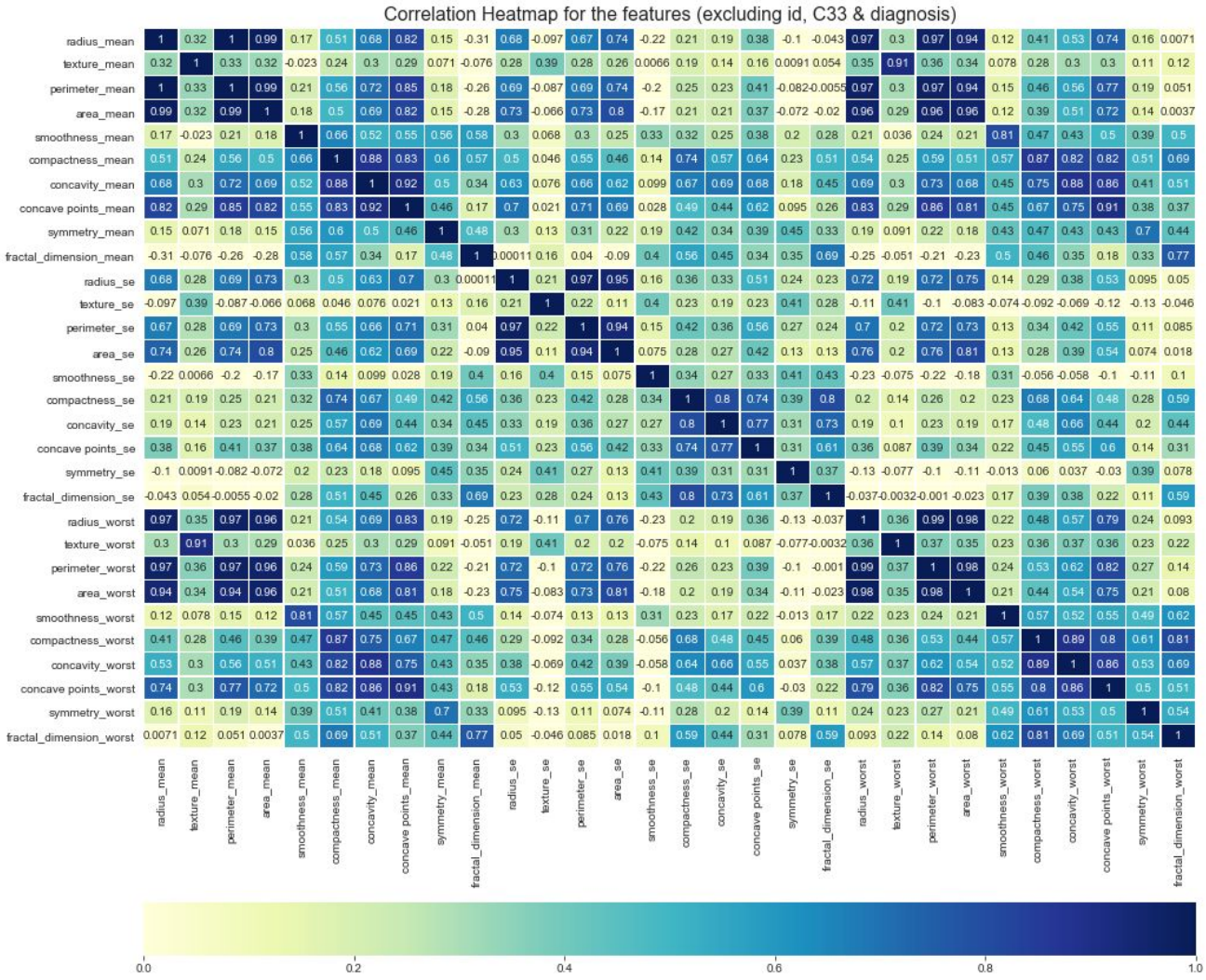
Below are some statistics for the 10 features that showed the mean of the real-valued features. It is important to note that that was a count of 569 for each variable, meaning there wasn't any missing data in our dataset. It is also important to note that the values of the means differ wildly, from 0.04 to 654. Outside of this issue of scaling, it is quite easy to work with such a clean and full dataset.

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	fractal_dimension_mean
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.161900	0.057700
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	0.061540
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	0.066120
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440

This table shows the statistics for the worst, or mean of the largest three values, features. It has a similar boon of being a complete dataset, but also suffers from the lack of scale like the above table.

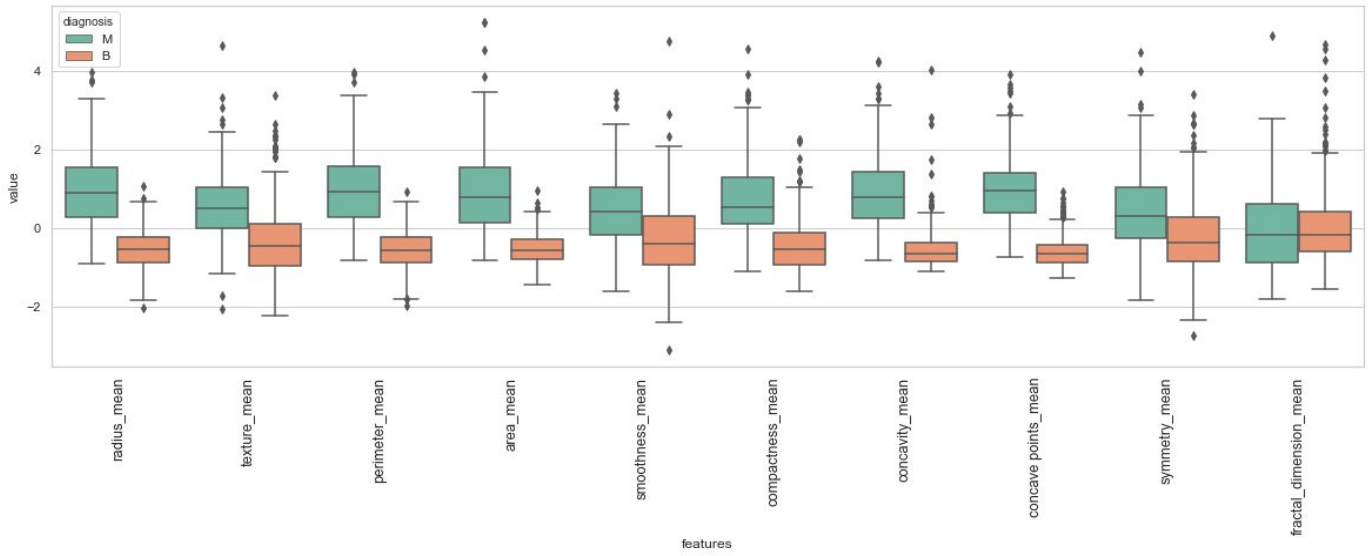
radius_worst	texture_worst	perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	fractal_dimension_worst
569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000
16.269190	25.677223	107.261213	880.583128	0.132369	0.254265	0.272188	0.114606	0.290076	0.083946
4.833242	6.146258	33.602542	569.356993	0.022832	0.157336	0.208624	0.065732	0.061867	0.018061
7.930000	12.020000	50.410000	185.200000	0.071170	0.027290	0.000000	0.000000	0.156500	0.055040
13.010000	21.080000	84.110000	515.300000	0.116600	0.147200	0.114500	0.064930	0.250400	0.071460
14.970000	25.410000	97.660000	686.500000	0.131300	0.211900	0.226700	0.099930	0.282200	0.080040
18.790000	29.720000	125.400000	1084.000000	0.146000	0.339100	0.382900	0.161400	0.317900	0.092080
36.040000	49.540000	251.200000	4254.000000	0.222600	1.058000	1.252000	0.291000	0.663800	0.207500

b. Correlation Matrix

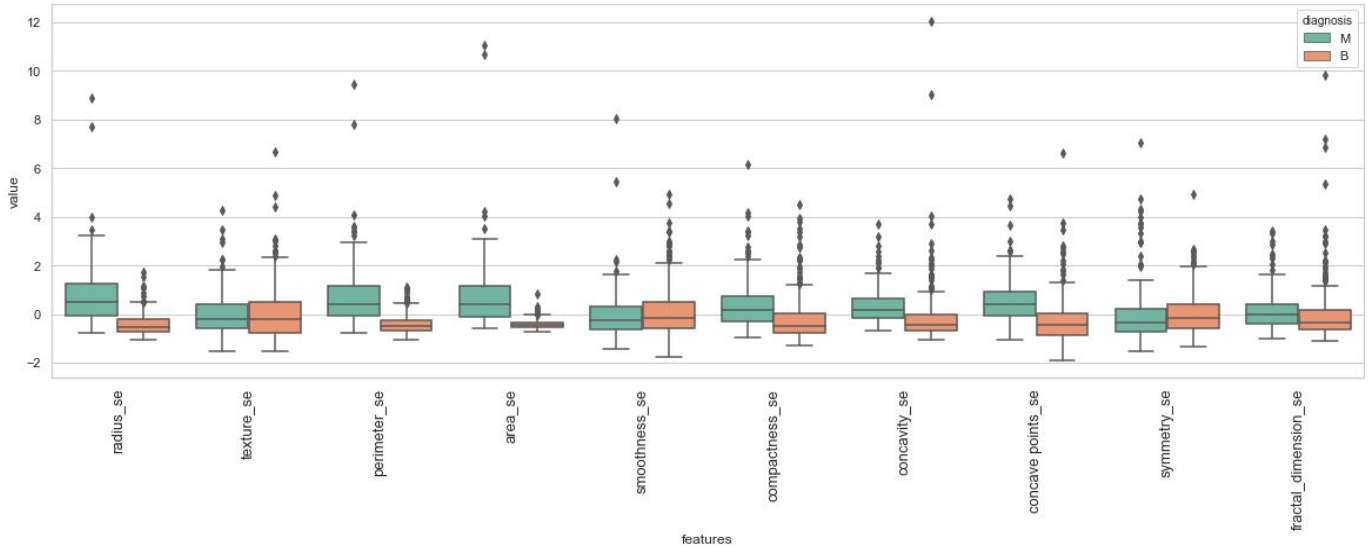


We used this correlation matrix to determine if there was any correlation between any of the columns. This would lead to a lack of model interpretability because of the correlations. We noticed that each column's mean, worst, and se were correlated, which makes sense intuitively -- they are using the same values. We also noticed that the values relating to size were also correlated, i.e. radius, perimeter, and area.

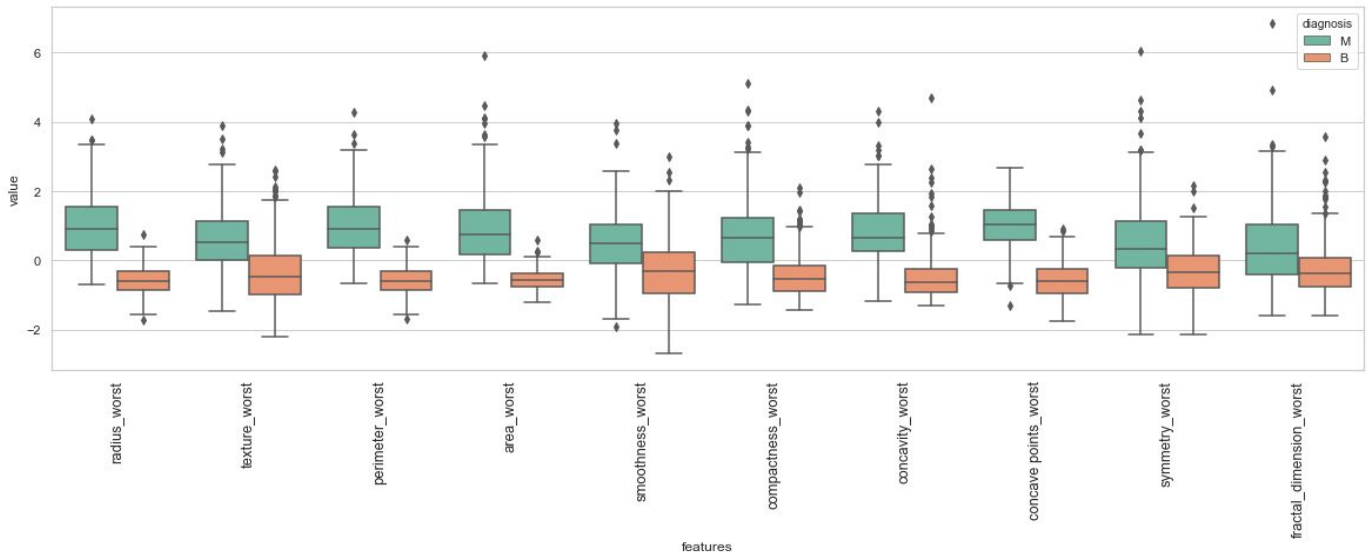
c. Boxplot



From the boxplots of the mean (divided by classification status), we can see that there are a few outliers for each column. Some columns, like perimeter, are more densely packed, while columns like fractal dimension have many more outliers. This makes sense, because fractal dimensions is an approximation and likely to be more varied.



We see similar patterns for the standard error of each of the values of the columns, except there are more outliers for each column. This also makes intuitive sense because the normal distribution bell curve is likely to be taller as values crowd around the mean. Having a smaller standard deviation is going to reveal more outliers, which is what we see from the above graph.



For our analysis of the three largest values, we expect that the boxplots for each column grow wider, as the three largest values are likely to create the most variance. This is exactly what we see in the above boxplots.

From the above three boxplots, and the meaning behind each variable, we believe that using the "mean" columns are likely to give us the best signals for determining whether a tumor is benign or malignant.

III. Literature Review

a. Sagemaker

Amazon SageMaker is a fully managed machine learning service designed to provide users with a simple, fast, and scalable way to perform typical machine learning project work-flow including build, train, and deployment.

The *build* step is done by connecting to other AWS services like S3 and transforming data in Amazon SageMaker Notebooks. It can be done with few clicks to create a Jupyter notebook instance with desirable size and capacity. And we can then perform data cleaning and exploration processes during the Jupyter hub running. The best feature here is that we will be able to choose a desired sever size for the notebook instance and shut down the instance after some period of inactivity to prevent unnecessary overhead.

The *training* step is about using AWS SageMaker algorithms and frameworks or using our algorithms and frameworks for distributed training.

In the *deploy* step, the model can be deployed to Amazon SageMaker endpoints for real-time or batch prediction. By specifying the required server capacity, we can deploy our machine learning model with just a single line of code. Use endpoint addresses to create application services or serverless functions.

There are several build-in algorithms for supervised learning:[4]

✓ Linear learner algorithm	● Factorization Machines Algorithm	✓ XGBoost Algorithm
● Object2Vec Algorithm	● DeepAR Forecasting Algorithm	● K-Nearest Neighbors (k-NN) Algorithm

b. H2O

H2O was created in 2014 but stabilized in 2017 as an open-source framework [2] and a Java-based software used to implement many machine learning libraries in memory. It provides distributed, parallel and memory processing for machine learning algorithms. The Enterprise Edition accommodates additional customization and support. In addition to using Java, users can also use R, Python, and other known languages to build models in H2O, or use H2O Flow (an interactive user interface based on a graphical notebook without any coding).

For supervised learning, H2O supports the following supervised algorithms:[3]

✓ AutoML	● Cox Proportional Hazards (CoxPH)	● Deep Learning (Neural Networks)	● Stacked Ensembles
● Naïve Bayes Classifier	● Distributed Random Forest (DRF)	● Generalized Linear Model (GLM)	● Support Vector Machine (SVM)
● RuleFit	● Generalized Additive Models (GAM)	✓ Gradient Boosting Machine (GBM)	● XGBoost (Not support in Window)

Remark that since XGBoost is not available in Windows, we will not be able to apply it due to the hardware limitation.

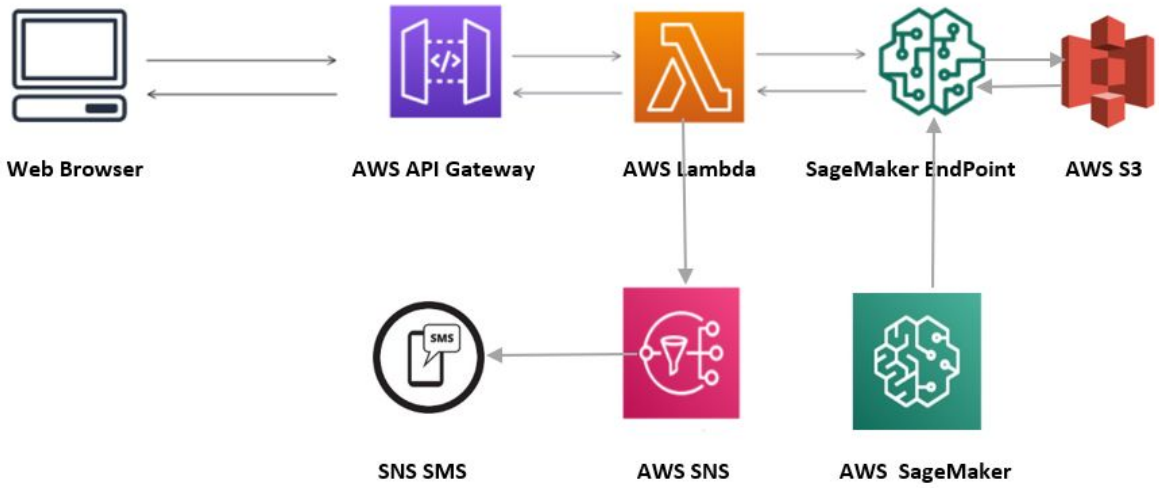
c. SparkML

Spark ML was built on top of Apache Spark and introduced by Spark 1.2 in 2017. It is open-source(https://github.com/apache/spark/tree/master/python/pyspark/ml) and mainly provides useful APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline/workflow.

For binary classification, Spark MLlib supports the following algorithms:[6]

✓ Linear SVMs	● Logistic Regression	● Decision Trees
● Gradient-Boosted Trees	● Naïve Bayes	● Random Forests

d. General Architecture of the Project



We plan on making use of both AWS Lambda and AWS API Gateway to allow users and clients to make predictions using our deployed model. AWS Lambda is an event-driven, serverless compute service that lets you run code without provisioning or managing servers, it provides flexibility to write custom logic and integrate with other aws services. AWS API gateway can expose an API that accepts parameters for our model to accept. AWS Lambda can parse these parameters and pass them to our model. The model will then return a prediction back to AWS Lambda, then AWS API Gateway, and, finally, back to the user [8].

Leveraging the architecture to deliver end to end user-centric feel, the ultimate prediction will be brought to the user-end point in the form of SMS using AWS SNS (Simple Notification Service). To bring it into service, the AWS lambda service will be extended with AWS SNS service by invoking customized triggers in lambda. Amazon Simple Notification Service (Amazon SNS) is a fully managed messaging service for both application-to-application (A2A) and application-to-person (A2P) communication. The A2P functionality enables to send messages to users at scale via SMS, mobile push, and email[9].

As our proposed architecture uses different AWS Services stack, it is required to integrate different services in a manner that one service can get access to another service. This is done by AWS IAM (Identity and Access Management) service. AWS IAM enables us to manage access to AWS services and resources securely. It is achieved by creating policies and attaching them to IAM identities (users, groups of users, or roles) or AWS resources[10].

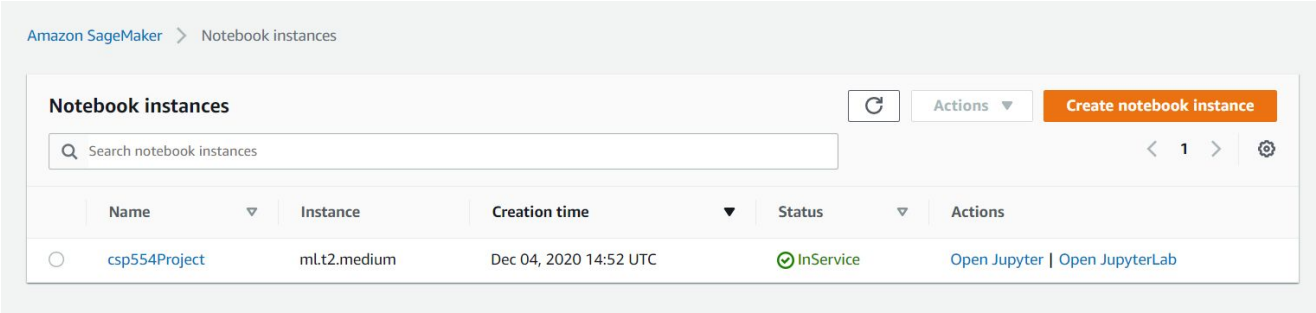
IV. Project Execution/Results

a. Sagemaker

- **Algorithm used: XGBoost**

The **XGBoost (eXtreme Gradient Boosting)** is a popular and efficient open-source implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm that attempts to accurately predict a target variable by combining an ensemble of estimates from a set of simpler and weaker models. The XGBoost algorithm performs well in machine learning competitions because of its robust handling of a variety of data types, relationships, distributions, and the variety of hyperparameters that you can fine-tune. You can use XGBoost for regression, classification (binary and multiclass), and ranking problems. We concentrated on the **classification** situation since we have a binary response variable called Diagnosis.

We used a **Jupyter Notebook** instance to execute our code for training and testing our XGBoost model.



- **Source Code:**

Importing the data, removing unnecessary columns, and checking for missing values.

```
[1]: import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import seaborn as sns
```

```
[2]: data = pd.read_csv("breast_cancer_data.csv")
```

[4]:
```
data.columns
```

[4]:
```
Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst', 'Unnamed: 32'],
      dtype='object')
```

[5]:
```
data = data.drop(['id'], axis = 1)
data = data.drop(['Unnamed: 32'], axis = 1)
```

[6]:
```
# Missing Value check
data.isnull().sum()
```

[6]:
```
diagnosis                0
radius_mean              0
texture_mean             0
perimeter_mean           0
area_mean                0
smoothness_mean          0
compactness_mean         0
concavity_mean           0
concave points_mean      0
symmetry_mean            0
fractal_dimension_mean   0
radius_se                0
texture_se               0
perimeter_se             0
area_se                  0
smoothness_se            0
compactness_se           0
concavity_se             0
concave points_se        0
symmetry_se              0
fractal_dimension_se     0
radius_worst             0
texture_worst            0
perimeter_worst          0
area_worst               0
smoothness_worst         0
compactness_worst        0
concavity_worst          0
concave points_worst     0
symmetry_worst           0
fractal_dimension_worst  0
dtype: int64
```

Importing necessary packages to setup the Sagemaker session and work with the algorithm.
Then an **IAM** role is defined via the ***get_exexution_role()*** function, a default S3 bucket is set up for storing the contents required for the model building and the model outputs.

The data is then divided into training and test sets and the training data is uploaded to the **S3 bucket**. The variable s3_input_train contains the path to the training data in the S3 bucket. This would be used as an input while training the model.

[30]:
```
import boto3, re, sys, math, json, os, sagemaker, urllib.request
from sagemaker import get_execution_role
from sagemaker.session import TrainingInput
from sagemaker.serializers import CSVSerializer
```

[9]:
```
# Defining IAM Role and setting up S3 bucket
role = get_execution_role()
prefix = 'sagemaker/breastcancer' # Folder to be created inside bucket
bucket_name = sagemaker.Session().default_bucket()
my_region = boto3.session.Session().region_name
print(my_region)
```

```
us-east-1
```

[10]:
```
bucket_name
```

[10]:
```
'sagemaker-us-east-1-844927434406'
```

[11]:
```
# Train-test split
train_data, test_data = np.split(data.sample(frac=1, random_state=123), [int(0.
 ↪8 * len(data))])
```

[61]:
```
train_data.diagnosis = pd.Categorical(pd.factorize(train_data.diagnosis)[0])
test_data.diagnosis = pd.Categorical(pd.factorize(test_data.diagnosis)[0])
```

[62]:
```
# AWS Sagemaker requires data to be in csv format without headers
train_data.to_csv('train.csv', index = False, header = False)
```

[63]:
```
# Uploading training data to S3 bucket
boto3.Session().resource('s3').Bucket(bucket_name).Object(os.path.join(prefix,␣
 ↪'train/train.csv')).upload_file('train.csv')

# Setting up training data input path for sagemaker
s3_input_train = TrainingInput('s3://{}/{}/train/'.format(bucket_name, prefix),␣
 ↪content_type='text/csv')
```

Setting up the **Sagemaker estimator**, model hyperparameters and training the model.
The XGBoost container is first retrieved using the ***image_uris()*** function and then used as the model image in the sagemaker estimator.

The model is then trained using the training data present in the **S3 bucket**. After successfully training the model, it is then deployed to create an endpoint for testing via the ***model.deploy()*** function. This endpoint would be used for prediction on new data.

[18]:
```
# Extreme Gradient Boosting
from sagemaker import image_uris

xgb_container = image_uris.retrieve("xgboost", my_region, "1.0-1")
sess = sagemaker.Session()
# Initialize the estimator
xgb = sagemaker.estimator.Estimator(image_uri = xgb_container,
                role=role,
                instance_count=1,
                instance_type='ml.m4.xlarge',
                output_path='s3://{}/{}/output'.format(bucket_name, prefix),
                sagemaker_session=sess)

# Set hyperparameters
xgb.set_hyperparameters(
        max_depth = 5,
        eta = 0.2,
        objective = "binary:logistic",
        num_round = 1000)
```

[64]:
```
# Training the model
# Ignore warning messages on output
xgb.fit({'train': s3_input_train})
```

```
2020-12-04 19:33:49 Starting - Starting the training job…
2020-12-04 19:33:51 Starting - Launching requested ML instances…
2020-12-04 19:35:06 Starting - Preparing the instances for training…


2020-12-04 19:37:11 Uploading - Uploading generated training model
2020-12-04 19:37:11 Completed - Training job completed
Training seconds: 71
Billable seconds: 71
```

[71]:
```
# Creating a model endpoint for testing
xgb_predictor = xgb.deploy(initial_instance_count=1, serializer =␣
 ↪CSVSerializer(), instance_type='ml.m4.xlarge')
```

```
----------------!
```

Testing the model on new data and analysis of performance.
The test data was converted to an array before using it as an input to the **predict()** function. The sagemaker outputs generated are in the form of comma separated text. Those were converted back to an array for the purpose of performance analysis.

We can see that the XGBoost model is performing well on the test data with an accuracy of **94.7%**.

[72]:
```
test_data_array = test_data.drop('diagnosis', axis = 1).values
```

[73]:
```
# Predictions are output as a string of probabilities separated by ','.
# That is converted to an array.
predictions = xgb_predictor.predict(test_data_array).decode('utf-8')
predictions_array = np.fromstring(predictions[1:], sep=',')
```

[75]:
```
# Probabilities to class labels
predictions_array = np.where(predictions_array > 0.5, 1, 0)
predictions_array
```

[75]:
```
array([0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1,
       0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0,
       1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0,
       0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1,
       1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,
       0, 0, 1, 0])
```

[85]:
```
from sklearn.metrics import accuracy_score, roc_curve, roc_auc_score
accuracy_score(test_data['diagnosis'], predictions_array)
```

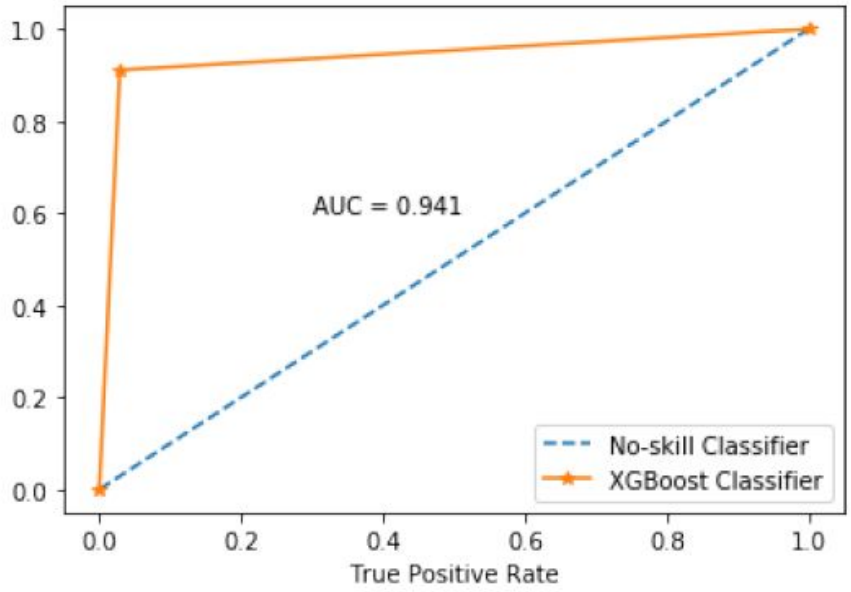
[85]:
```
0.9473684210526315
```

Additionally, we plotted an **ROC curve**. It is evident that the classifier is very good in terms of precision and recall as well. The Area Under the Curve is big enough for us to conclude that **this is an appropriate classifier for our job**.

[86]:
```
# Calculating the roc_curve parameters - TPR, FPR, Area Under the Curve (AUC)
ns_labels = [0 for _ in range(len(test_data['diagnosis']))]
ns_fpr, ns_tpr, _ = roc_curve(test_data['diagnosis'], ns_labels)
xgb_fpr, xgb_tpr, _ = roc_curve(test_data['diagnosis'], predictions_array)
AUC = roc_auc_score(test_data['diagnosis'], predictions_array)
```

[89]:
```
# Analysis of ROC curve
plt.plot(ns_fpr, ns_tpr, linestyle = '--', label = "No-skill Classifier")
plt.plot(fpr, tpr, marker = '*', label = "XGBoost Classifier")
plt.xlabel('False Positive Rate')
plt.xlabel('True Positive Rate')
plt.text(0.3, 0.6, "AUC = {}".format(np.round(AUC,3)))
plt.legend()
```

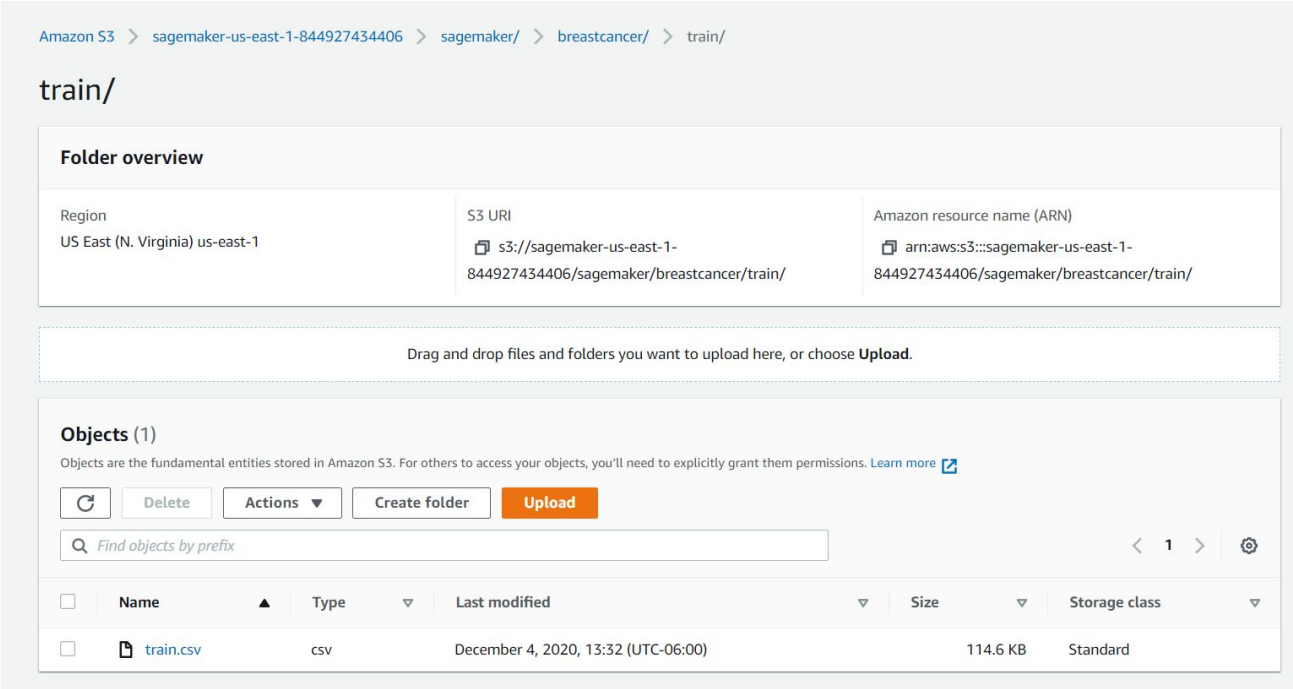
[89]: <matplotlib.legend.Legend at 0x7fcb17dfbba8>



[AWS Sagemaker Screenshots]

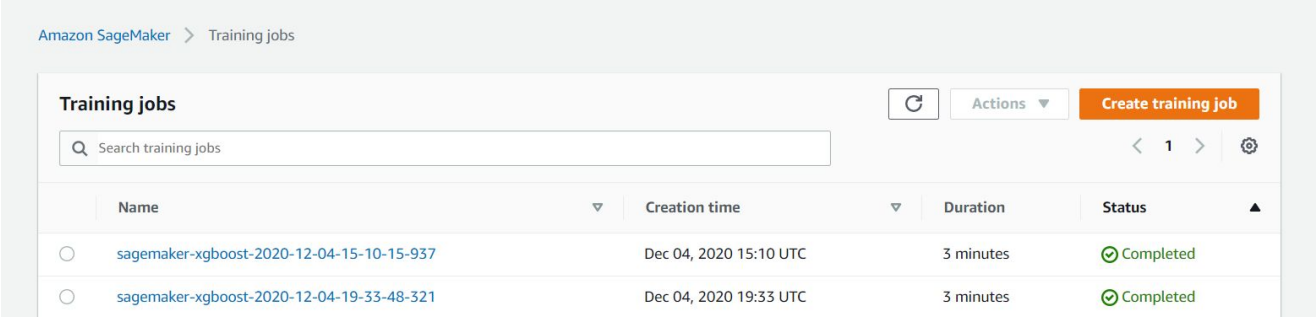
i. Splitted Training Data on S3:

Using SageMaker we stored our training and testing data in S3. This allowed us to access the data from many different AWS services and allowed us to store the data outside of our SageMaker instance's local storage.



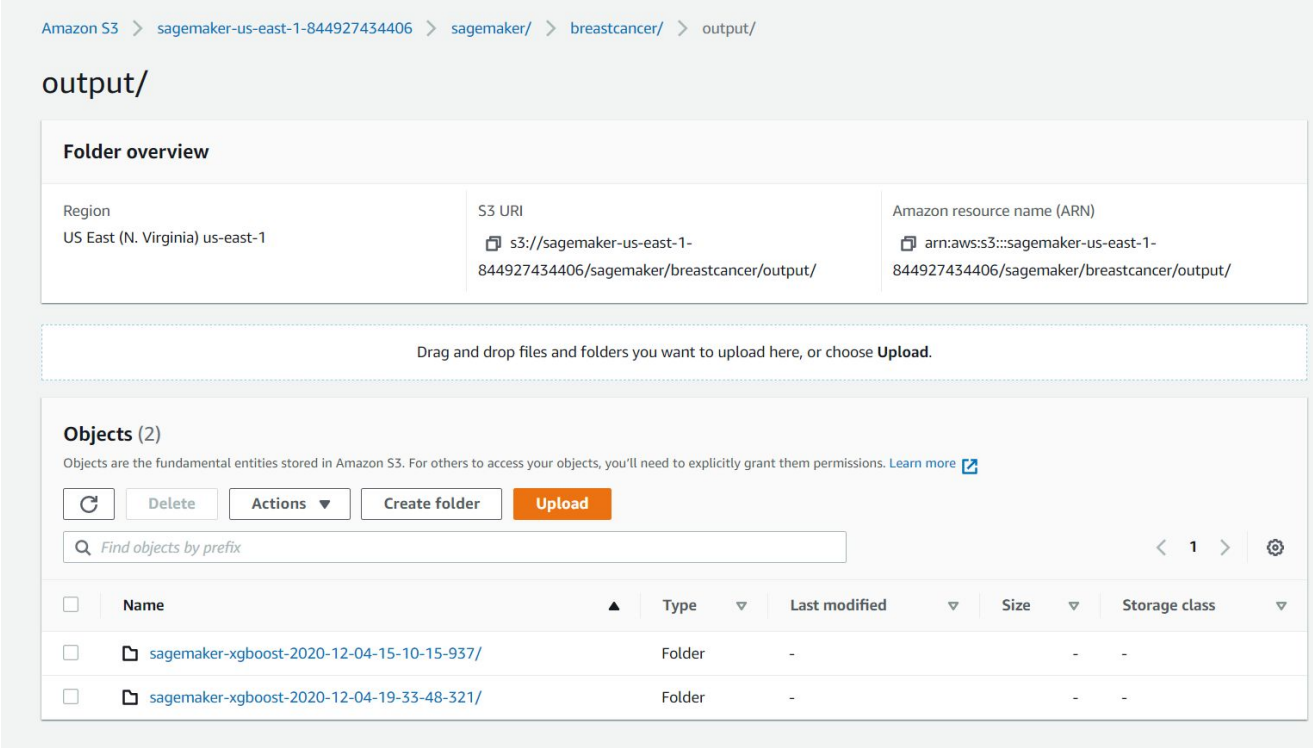
ii. Training Job Completion Track on SageMaker:

Whenever we completed a job we navigated to SageMaker's training jobs tab. This allowed us to monitor the status and duration of each of our training jobs.



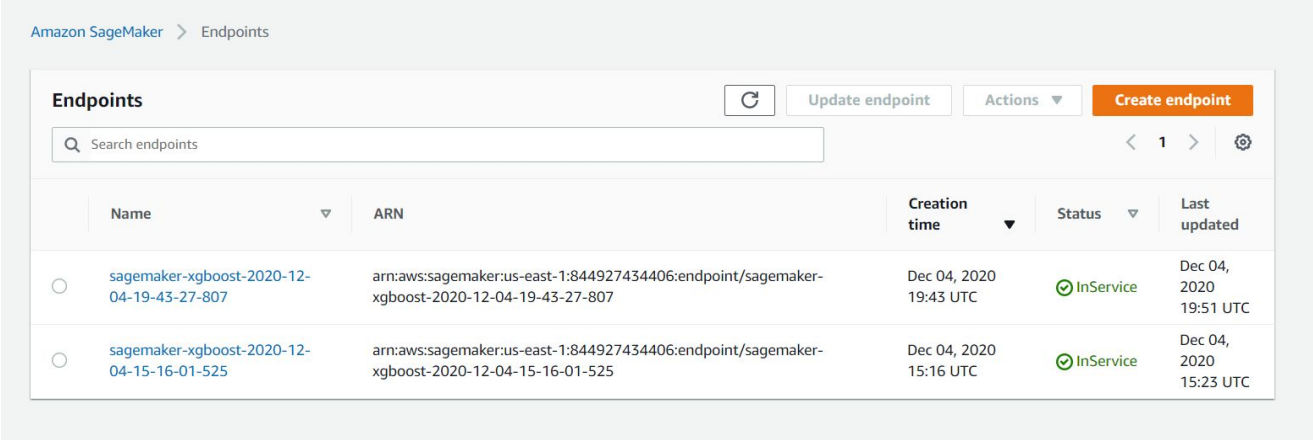
iii. Train Model stored S3 Bucket:

In addition to storing our data in S3, we also stored our model in an S3 bucket. In the below picture, we see our xg boost model in different folders, one for each version that we created.



iv. SageMaker EndPoint Generation:

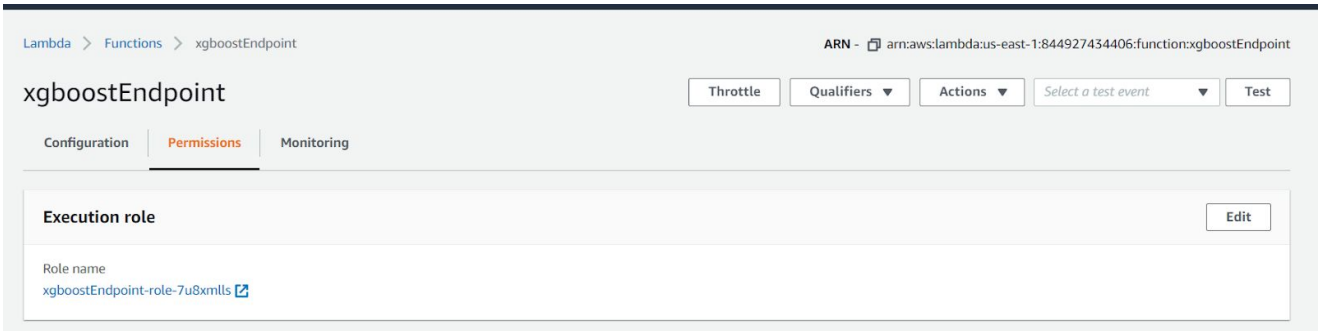
For each of the models we created above, we also created model endpoints using SageMaker's endpoint functionality. This allows us to use AWS Lambda, which we describe in more detail in the following sections.



b. AWS Lambda:

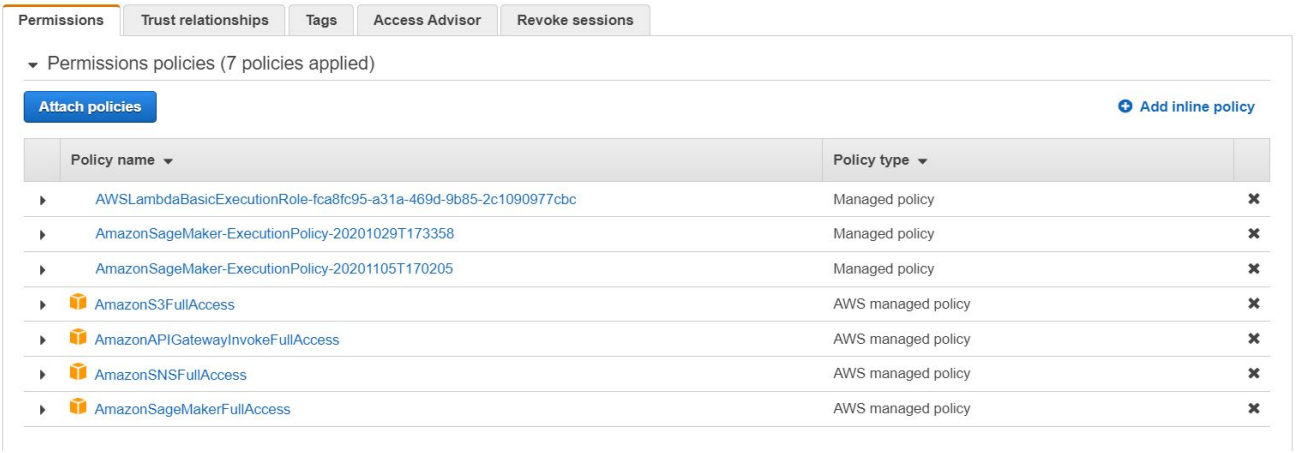
i. AWS Lambda Function Rule

We have created a lambda function: xgboostEndpoint. For that the AWS IAM rule along with attached policies are required for the role. The role will automatically be generated while creating the AWS Lambda function which is shown in below figure.



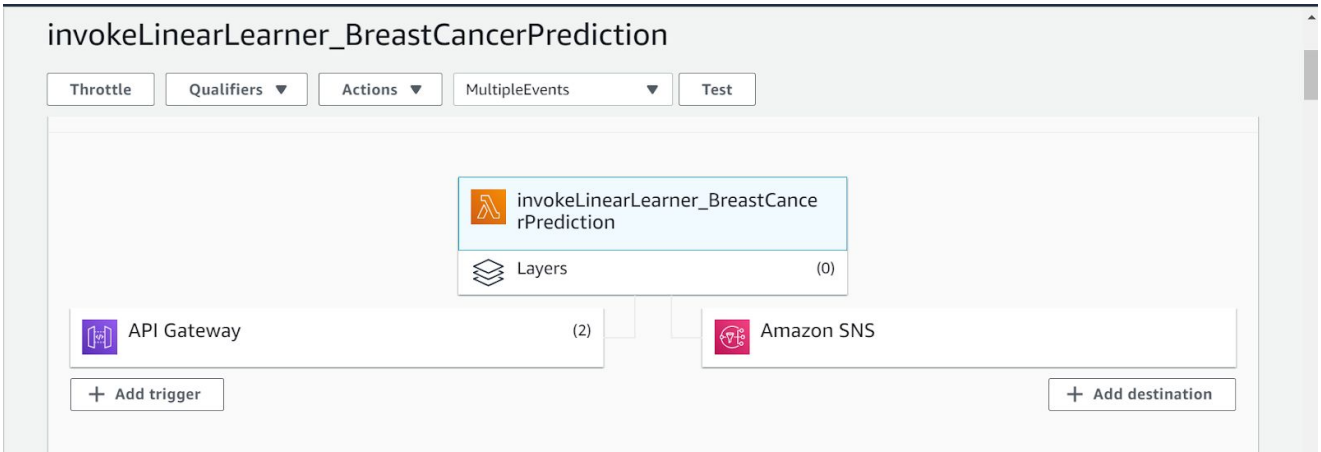
ii. AWS Lambda Rule IAM Access Policies

Here for the Lambda rule, attached IAM policies screenshot is given below, as we required Lambda to access S3, SNS, API gateway, sagemaker end point.



iii. AWS Lambda Integration:

Below figure depicts the integration of Lambda function with AWS API Gateway and AWS SNS, those parts are explained in further parts of the report.



iv. AWS Lambda Function Code:

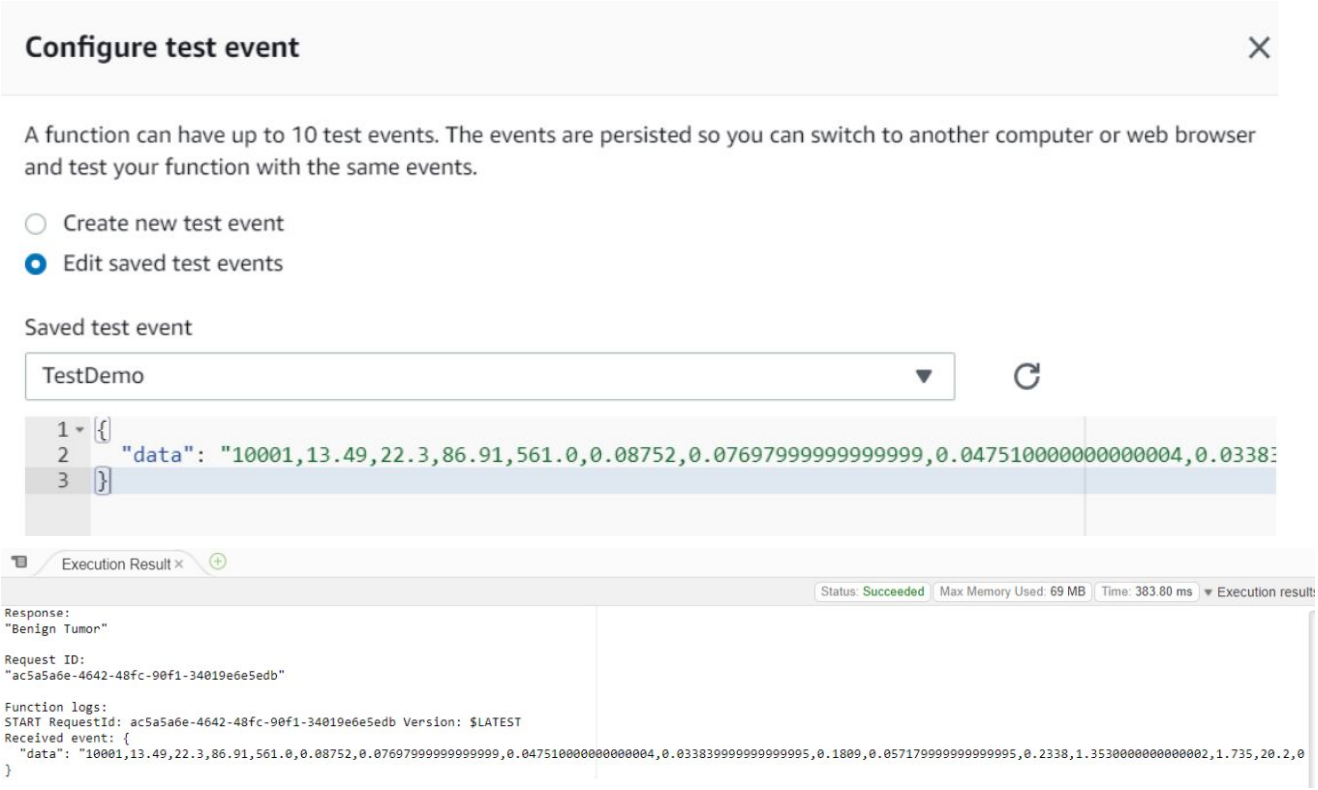
As a coding environment, we are using python 2.7 for our Lambda function. Below is the code we created to work with SageMaker endpoint, sns published sms service.

```
# grab environment variables
ENDPOINT_NAME = os.environ['ENDPOINT_NAME']
runtime= boto3.client('runtime.sagemaker')
sns = boto3.client('sns')

def lambda_handler(event, context):

    data = json.loads(json.dumps(event))
    payload = data['data']
    li = list(payload.split(","))
    str2 = str(li[0])
    str2_len = len(str2)
    payload = payload[str2_len + 1: len(payload)]
    str1 =''
    response = runtime.invoke_endpoint(EndpointName=ENDPOINT_NAME,
                                       ContentType='text/csv',
                                       Body=payload)

    print(response)
    result = json.loads(response['Body'].read().decode())
    pred = int(result['predictions'][0]['score'])
    predicted_label = 'Malignant Tumor' if pred == 1 else 'Benign Tumor'
    response_sns = sns.publish(
    TopicArn='arn:aws:sns:us-east-1:869509803755:BreastCancer_Topic',
    Message='User ='+str(li[0])+',The Prediction of model is :'+predicted_label, Subject = 'Breas
    return predicted_label
```

To iterate through the code above, here we used aws boto3 client used as SDK for python to communicate with other services, here runtime sagemaker endpoint and sns services.
Line 14-20 depicts how the data are input and manipulated with some string operations. Line 21-26 exhibits, runtime sagemaker endpoint invocation with payload and getting json response. As our model gives a response in the form of a floating digit. So, we converted into an integer and determined the value equals to 1 predict “Benign Tumor”, otherwise “Malignant Tumour”, depicted in line 27-28. Line 29-32 shows integration of lambda using sns topic and sends the trigger message at the client end-point.

v. Lambda Test Event and Output:

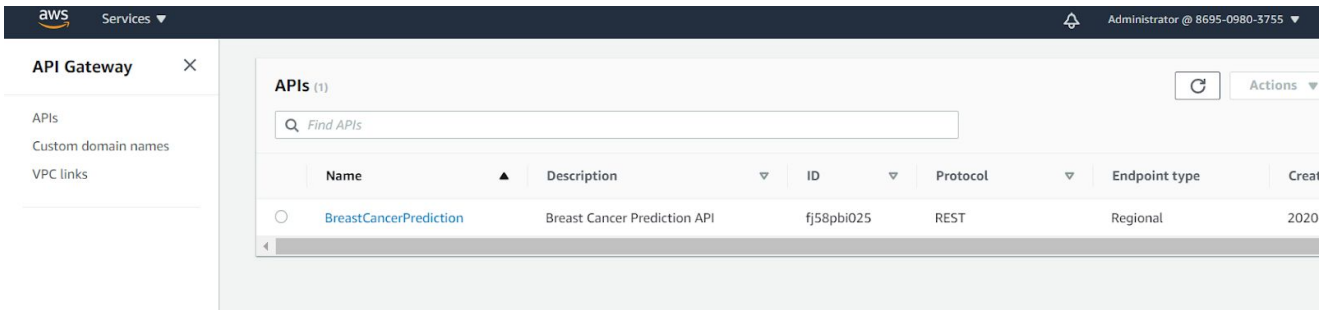
For testing purposes, we configured the test event called “TestDemo” and checked our lambda function.
In the execution result window, the output of the lambda function has been checked with logs, which are stored in AWS cloudwatch for further testing.



c. AWS API Gateway:

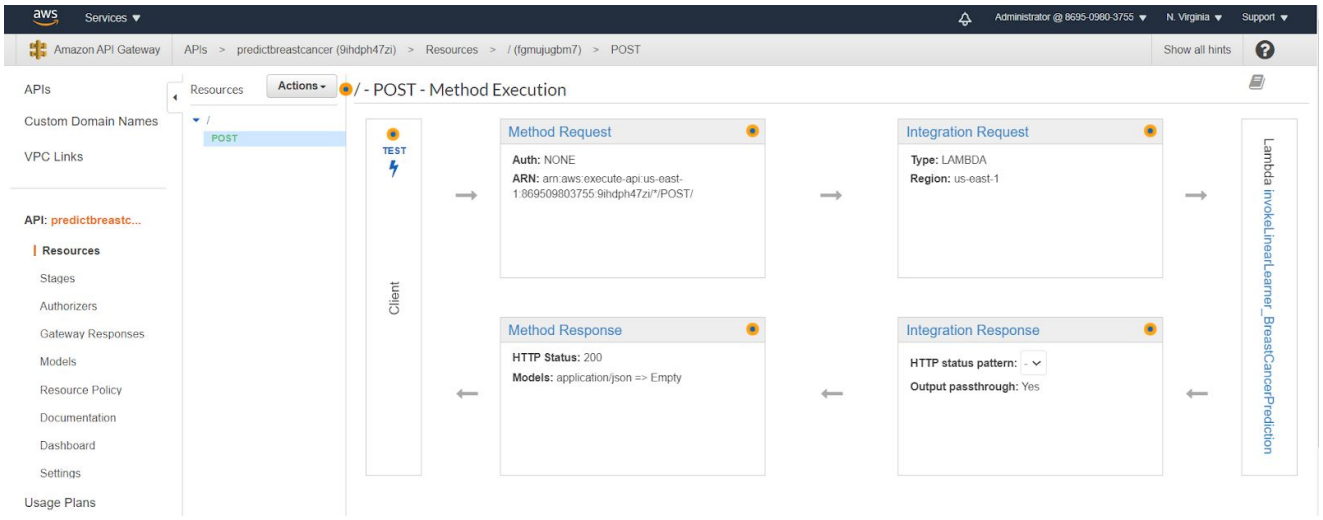
i. AWS API Created

To run the entire application for the client endpoint, we have used AWS API gateway which integrates with AWS lambda function explicitly. Here we have created an API called “BreastCancerPrediction”.



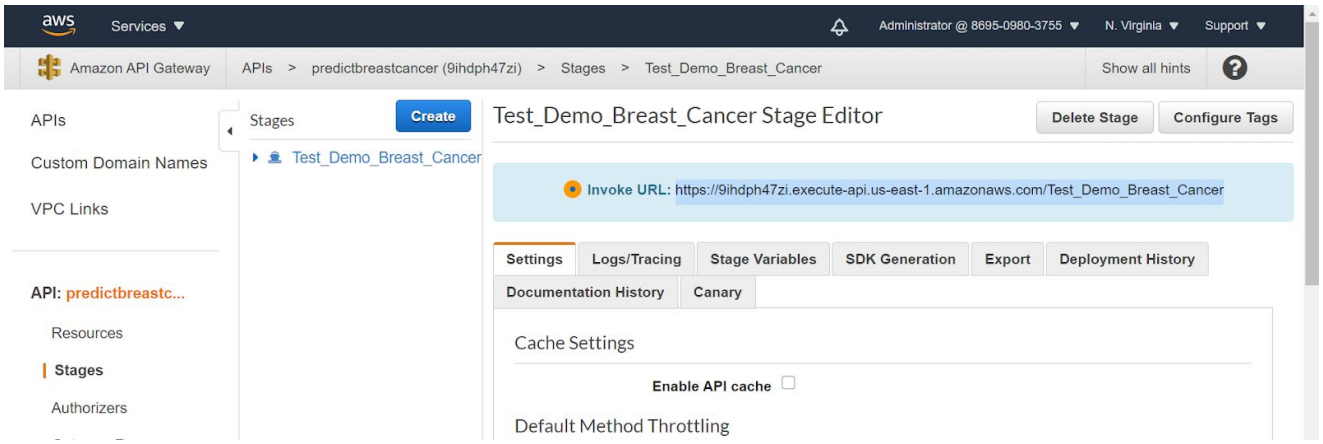
ii. API configured with AWS Lambda:

In API we have used resources as a post method for invoking client requests and added a lambda function, which we have created earlier (xgboostEndpoint).



iii. Invoked API URL:

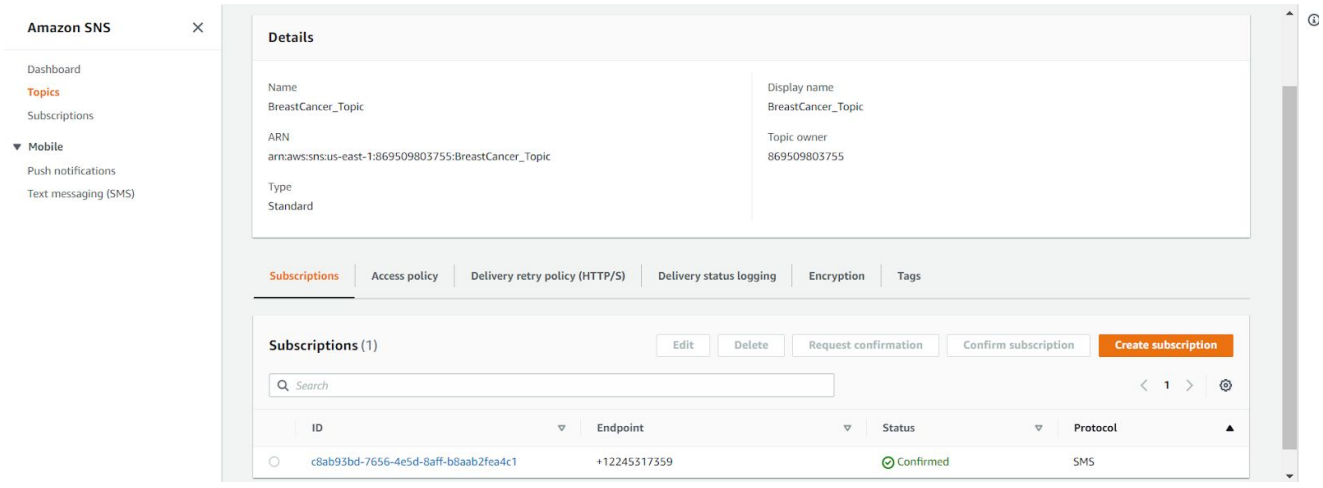
In stages of API, we can see generated API URL, which we are using further in our project for post method invocation from the client end. So, all that client needs should be API URL.



d. AWS SNS :

i. SNS Topic Created:

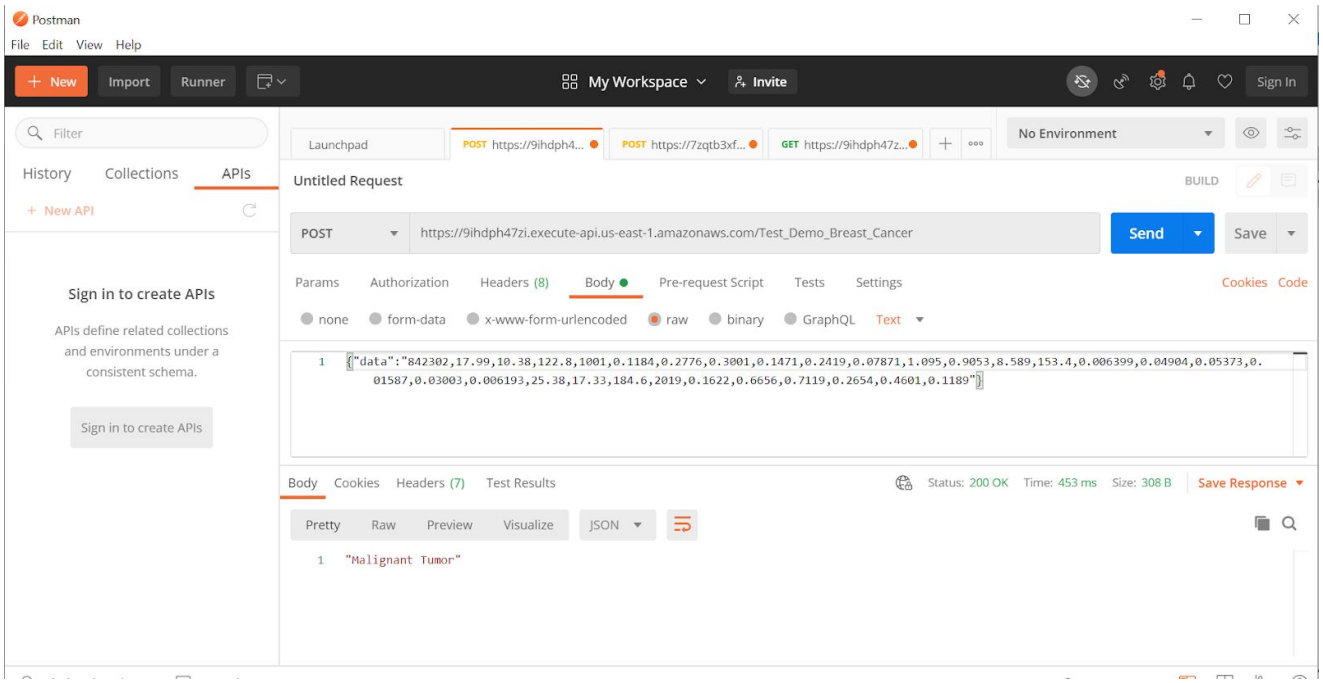
As a part of the final outcome, the prediction should be sent by message to the client using SNS service. SNS has different options like Text message, email. Here to consume service, we have chosen SNS push SMS and here we can see SNS ARN (Amazon Resource Name), that we have already used in our lambda code to integrate lambda with SNS.



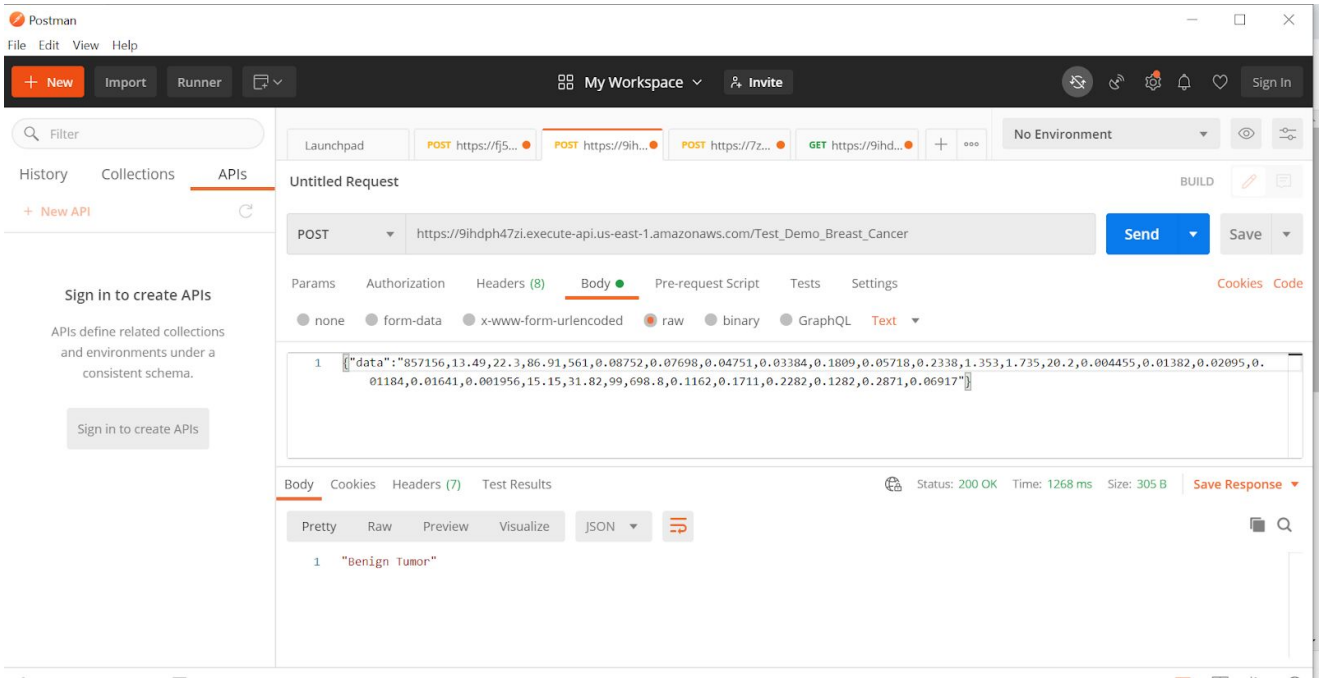
e. Postman Testing for API URL:

For final testing we used a postman tool to examine our API URL which we have generated using the API gateway. Here we show some test inputs and it's prediction via post method over API Url.

i. Input 1: Malignant Prediction

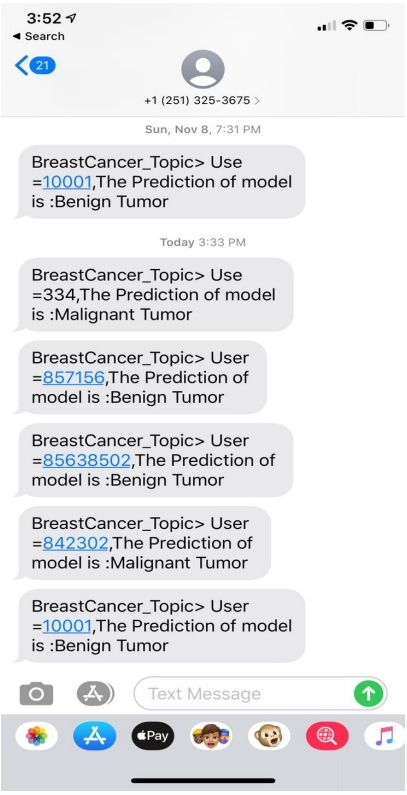


ii. Input 2: Benign Prediction



f. SNS Topic SMS Service Output:

As a final end-to-end feel, we used SNS Service, which we have tested and below is a screenshot for the prediction message we got when the serverless lambda trigger with Sagemaker runtime endpoint.



g. *Supplemental* - H20

Although in this project, our goal is mainly to focus on the implementation of Sagemaker, with the intention of exposing to more different big data machine learning technologies, we also tried the H2O in this project. Though here we only demonstrate analyzing the data on the H2O cluster, by creating an AWS Lambda deployment package from the model, we can then deploy a RESTful endpoint, and migrate our H2O Flows to AWS. [11]

i. Initialize H2O

We launched a Jupyter Notebook to run H2O Python commands.

```
>> h2o.init()
```

```
Checking whether there is an H2O instance running at http://localhost:54321 ..... not found.
Attempting to start a local H2O server...
; Java HotSpot(TM) 64-Bit Server VM (build 25.251-b08, mixed mode)
  Starting server from C:\Users\User\anaconda3\lib\site-packages\h2o\backend\bin\h2o.jar
  Ice root: C:\Users\User\AppData\Local\Temp\tmpulaq66qc
  JVM stdout: C:\Users\User\AppData\Local\Temp\tmpulaq66qc\h2o_User_started_from_python.out
  JVM stderr: C:\Users\User\AppData\Local\Temp\tmpulaq66qc\h2o_User_started_from_python.err
  Server is running at http://127.0.0.1:54321
Connecting to H2O server at http://127.0.0.1:54321 ... successful.
```

H2O_cluster_uptime:	03 secs
H2O_cluster_timezone:	America/Chicago
H2O_data_parsing_timezone:	UTC
H2O_cluster_version:	3.30.1.3
H2O_cluster_version_age:	1 month and 30 days
H2O_cluster_name:	H2O_from_python_User_5rwlg4
H2O_cluster_total_nodes:	1
H2O_cluster_free_memory:	4.405 Gb
H2O_cluster_total_cores:	0
H2O_cluster_allowed_cores:	0
H2O_cluster_status:	accepting new members, healthy
H2O_connection_url:	http://127.0.0.1:54321
H2O_connection_proxy:	{"http": null, "https": null}
H2O_internal_security:	False
H2O_API_Extensions:	Amazon S3, Algos, AutoML, Core V3, TargetEncoder, Core V4
Python_version:	3.7.6 final

In this step, H2O will first check if there is an existing instance, if there is one, H2O will try to connect to that one. If not, a new instance will be started. Pieces of information about this engine will be printed.

ii. Import the data

```
>> data_df = h2o.import_file("wbc_data.csv", destination_frame="data_df")
```

```
Parse progress: |████████████████████████████████████████████████████████| 100%
```

iii. Split & Convert the data.

We splitted the data into training/validation/test set in ratio 60/20/20. And convert the target column to factors as response for the following analysis.

```
>> train_df, valid_df, test_df = data_df.split_frame(ratios=[0.6,0.2], seed=2018)
>> train_df[target] = train_df[target].asfactor()
>> valid_df[target] = valid_df[target].asfactor()
>> test_df[target] = test_df[target].asfactor()
```

```
Number of rows in train, valid and test set :  344 124 101
```

As the result shown above, we can see that the length for training/validation/test set is 344/124/101 respectively.

iv. Train the GBM model

We first choose GBM (h2o-gbm-tutorials) as our first target algorithm. Conceptually, GBM constructs a positive stepwise additive model by implementing gradient descent in the function space.

To train a GBM in H2O, we need to first initialize a H2O GBM estimator with a seed setting. Then we can train our data with this initialized GBM model.

```
>> gbm = H2OGradientBoostingEstimator(seed=1122)
>> gbm.train(x=predictors, y=target, training_frame=train_df)
```

```
gbm Model Build progress: |████████████████████████████████████████| 100%
```

```
Model Summary:
```

	number_of_trees	number_of_internal_trees	model_size_in_bytes	min_depth	max_depth	mean_depth	min_leaves	max_leaves	mean_leaves
0	50.0	50.0	9917.0	4.0	5.0	4.98	7.0	14.0	11.18

v. Model Performance

- **For the Training set**

```
>> gbm.model_performance(train_df)
```

```
ModelMetricsBinomial: gbm
** Reported on test data. **

MSE: 0.0006435058034842673
RMSE: 0.025367416176746645
LogLoss: 0.011979760052185713
Mean Per-Class Error: 0.0
AUC: 1.0
AUCPR: 1.0
Gini: 1.0

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.7005413230437432:
```

		B	M	Error	Rate
0	B	215.0	0.0	0.0	(0.0/215.0)
1	M	0.0	129.0	0.0	(0.0/129.0)
2	Total	215.0	129.0	0.0	(0.0/344.0)

- **For the Validation set**

```
>> gbm.model_performance(valid_df)
```

```
ModelMetricsBinomial: gbm
** Reported on test data. **

MSE: 0.013297212568117988
RMSE: 0.11531354026357003
LogLoss: 0.05053516736848985
Mean Per-Class Error: 0.012820512820512775
AUC: 0.9987933634992459
AUCPR: 0.9975866156502365
Gini: 0.9975867269984917

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.5851063290701174:
```

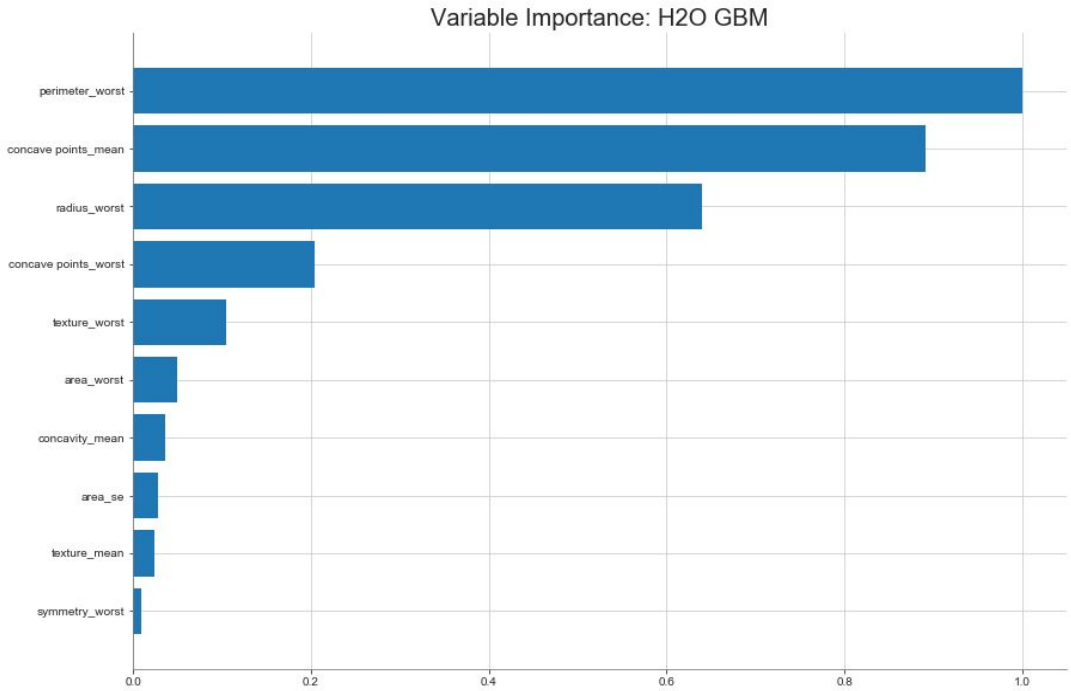
		B	M	Error	Rate
0	B	85.0	0.0	0.0	(0.0/85.0)
1	M	1.0	38.0	0.0256	(1.0/39.0)
2	Total	86.0	38.0	0.0081	(1.0/124.0)

Since the model performed quite well in the validation set, we will not need to further tune the model with different parameters, and can now do the model prediction using the test set predictors.

- **Important variables**

Show important variables for the classification.

```
>> gbm.varimp_plot()
```



As the result shown above, we can see that the features *perimeter_worst*, *concave point_means*, and *redius_worst* are relatively more important variables for breast cancer classification than other variables.

vi. Model Prediction

```
>> gbm.predict(test_df[predictors])
```

```
gbm prediction progress: |████████████████████████████████████████████████| 100%
```

predict	B	M
M	0.00531407	0.994686
M	0.17262	0.82738
B	0.996533	0.00346675
M	0.00504308	0.994957
B	0.992035	0.00796517
B	0.887805	0.112195
B	0.982393	0.0176073
B	0.674906	0.325094
B	0.996696	0.00330358
B	0.996922	0.0030781

vii. AutoML : Automatic Machine Learning

We also tried another algorithm, AutoML(h2o-automl-tutorial), to do the breast cancer diagnosis classification. H2O's AutoML can be used for automating the machine learning workflow, which includes automatic training and tuning of many models within a user-specified time-limit.

```
>> from h2o.automl import H2OAutoML
>> aml = H2OAutoML(max_models = 10, max_runtime_secs=100, seed = 1122)
>> aml.train(x=predictors, y=response, training_frame=train, validation_frame=valid)
```

```
AutoML progress: |
00:15:38.899: User specified a validation frame with cross-validation still enabled. Please note that the models will still be validated using cross-validation only, the validation frame will be used to provide purely informative validation metrics on the trained models.
00:15:38.900: AutoML: XGBoost is not available; skipping it.

████████████████████████████████████████████████| 100%
```

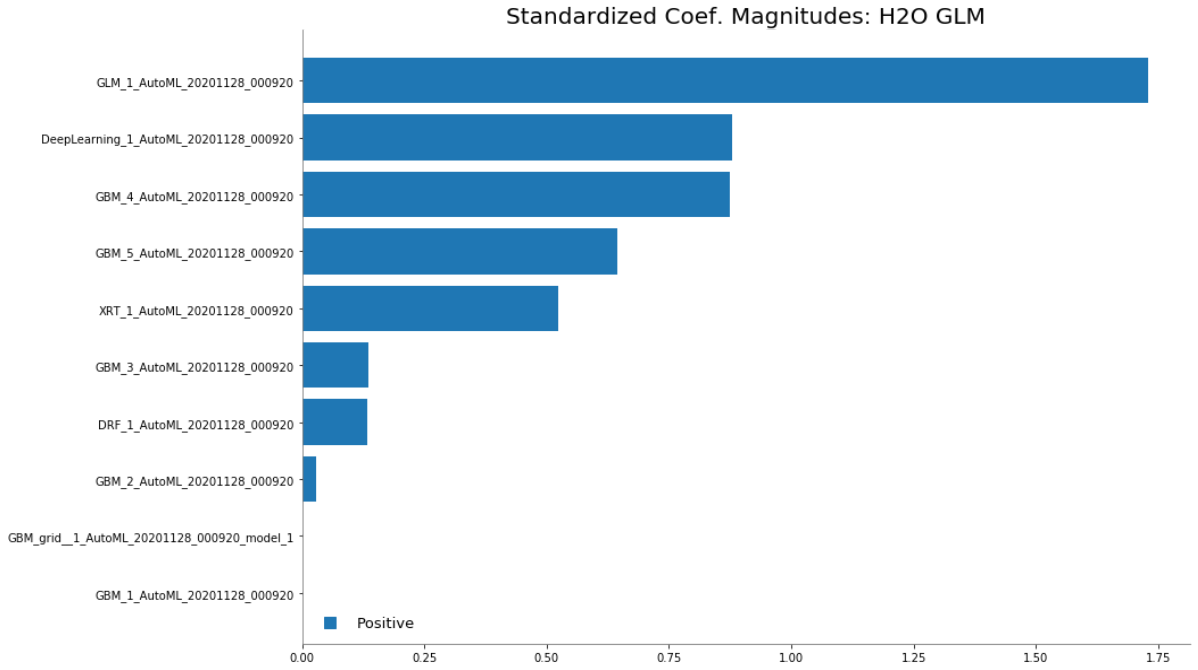
- **AutoML Leaderboard**

```
>> aml.leaderboard
```

model_id	auc	logloss	aucpr	mean_per_class_error	rmse	mse
GLM_1_AutoML_20201128_000920	0.998224	0.0599249	0.996901	0.014985	0.126007	0.0158777
StackedEnsemble_BestOfFamily_AutoML_20201128_000920	0.997891	0.0580989	0.996201	0.017205	0.1254	0.0157251
StackedEnsemble_AllModels_AutoML_20201128_000920	0.997595	0.0614578	0.995634	0.021534	0.130041	0.0169107
GBM_4_AutoML_20201128_000920	0.997558	0.0642939	0.995495	0.0278055	0.136684	0.0186825
GBM_grid__1_AutoML_20201128_000920_model_1	0.995375	0.100105	0.991316	0.0407925	0.176293	0.0310793
XRT_1_AutoML_20201128_000920	0.994949	0.0972034	0.990399	0.042846	0.170333	0.0290134
GBM_3_AutoML_20201128_000920	0.994487	0.100397	0.989583	0.0385725	0.166939	0.0278687
DRF_1_AutoML_20201128_000920	0.994191	0.102683	0.989148	0.03441	0.17321	0.0300015
GBM_2_AutoML_20201128_000920	0.994043	0.122392	0.989272	0.0407925	0.173771	0.0301965
DeepLearning_1_AutoML_20201128_000920	0.992859	0.101924	0.987039	0.044955	0.166309	0.0276588

- Check the **contribution** of the individual models for this meta learner

```
model_ids = list(aml.leaderboard['model_id'].as_data_frame().iloc[:,0])
se = h2o.get_model([mid for mid in model_ids if "StackedEnsemble_AllModels" in mid][0])
metalearner = h2o.get_model(se.metalearner()['name'])
metalearner.std_coef_plot()
```



In this case, we can say that the GLM is the topmost contributor to the ensemble followed by the Geep Learning and GBM model.

viii. Shutdown the H2O

```
>> h2o.shutdown()
```

V. References

[1] L. D. P Cuong, Wang Dong, D. T. Hoang, L. M. N Uyen. Breast Cancer Prediction based on Deep Neural Network Model Implemented AWS Machine Learning Platform. International Journal of Recent Technology and Engineering (IJRTE) ISSN: 2277-3878, Volume-9 Issue-2, July 2020

[2] H2O - Open Source Leader in AI and ML - https://www.h2o.ai/products/h2o/

[3] Classification and Regression with H2O Deep Learning - https://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html

[4] Amazon SageMaker - https://aws.amazon.com/sagemaker/

[5] Spark ML Programming Guide - https://spark.apache.org/docs/1.2.2/ml-guide.html

[6] Machine Learning Library (MLlib) Guide - https://spark.apache.org/docs/latest/ml-guide.html#machine-learning-library-mllib-guide

[7] Kwon H, Park J, Lee Y. Stacking Ensemble Technique for Classifying Breast Cancer. Healthcare Informatics Research. 2019 Oct;25(4):283-288. DOI: 10.4258/hir.2019.25.4.283.

[8] Olsen, Rumi. Call an Amazon Sagemaker model endpoint using Amazon API Gateway and AWS Lambda. https://aws.amazon.com/blogs/machine-learning/call-an-amazon-sagemaker-model-endpoint-using-amazon-api-gateway-and-aws-lambda/. AWS Machine Learning Blog. 2019, Jul 19.

[9] AWS SNS Service - https://aws.amazon.com/sns/

[10] AWS Iam Service - https://aws.amazon.com/iam/

[11] 4 Steps to Train and Deploy Machine Learning Models on AWS Using H2O - https://aws.amazon.com/blogs/apn/4-steps-to-train-and-deploy-machine-learning-models-on-aws-using-h2o/

- Some kernels and existing sample code available in the following links:

[11] (Kaggle) Feature Selection and Data Visualization - https://www.kaggle.com/kanncaa1/feature-selection-and-data-visualization

[12] AWS Sagemaker XGBoost: https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html