Breast Cancer Prediction

Predict Breast Cancer using SageMaker's Linear-Learner with features derived from images of Breast Mass

---

Contents

1. Background
2. Setup
3. Data
4. Train
5. Host
6. Predict
7. Extensions

Background

This notebook illustrates how one can use SageMaker's algorithms for solving applications which require `linear models` for prediction. For this illustration, we have taken an example for breast cancer prediction using UCI'S breast cancer diagnostic data set. The purpose here is to use this data set to build a predictve model of whether a breast mass image indicates benign or malignant tumor. The data set will be used to illustrate:

- Basic setup for using SageMaker.
- Converting datasets to protobuf format used by the Amazon SageMaker algorithms and uploading to S3.
- Training SageMaker's linear learner on the data set.
- Hosting the trained model.
- Scoring using the trained model.

Setup

Let's start by specifying:

- The SageMaker role arn used to give learning and hosting access to your data. The snippet below will use the same role used by your SageMaker notebook instance, if you're using other. Otherwise, specify the full ARN of a role with the SageMakerFullAccess policy attached.
- The S3 bucket that you want to use for training and storing model objects.

In [57]:
```
import os
import boto3
import re
import sagemaker

role = sagemaker.get_execution_role()
region = boto3.Session().region_name

# S3 bucket for saving code and model artifacts.
# Feel free to specify a different bucket and prefix
bucket = "day07-mk1"

prefix = (
    "sagemaker/DEMO-breast-cancer-prediction"  # place to upload training files within the bucket
)
print (region)
```

```
sagemaker.config INFO - Not applying SDK defaults from location: /etc/xdg/sagemaker/config.yaml
sagemaker.config INFO - Not applying SDK defaults from location: /home/ec2-user/.config/sagemaker/config.yaml
us-east-1
```

In [60]:
```
role
```

Out[60]:
```
'arn:aws:iam::585522057818:role/fast-ai-academic-11-Student-Azure'
```

Now we'll import the Python libraries we'll need.

In [3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import io
import time
import json
import sagemaker.amazon.common as smac
```

Data sources

> Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

> Breast Cancer Wisconsin (Diagnostic) Data Set [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)].

> *Also see:* Breast Cancer Wisconsin (Diagnostic) Data Set [https://www.kaggle.com/uciml/breast-cancer-wisconsin-data].

Let's download the data and save it in the local folder with the name data.csv and take a look at it.

In [4]:
```
s3 = boto3.client("s3")

filename = "wdbc.csv"
s3.download_file("sagemaker-sample-files", "datasets/tabular/breast_cancer/wdbc.csv", filename)
data = pd.read_csv(filename, header=None)

# specify columns extracted from wbdc.names
data.columns = [
    "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",
]

# save the data
data.to_csv("data.csv", sep=",", index=False)

# print the shape of the data file
print(data.shape)

# show the top few rows
display(data.head())

# describe the data object
display(data.describe())

# we will also summarize the categorical field diganosis
display(data.diagnosis.value_counts())
```

```
(569, 32)
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_m
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.

5 rows × 32 columns

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_me
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.0000
mean	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.0887
std	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.0797
min	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.0000
25%	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.0295
50%	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.0615
75%	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.1307
max	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.4268

8 rows × 31 columns

```
diagnosis
B    357
M    212
Name: count, dtype: int64
```

Key observations:

- Data has 569 observations and 32 columns.
- First field is 'id'.
- Second field, 'diagnosis', is an indicator of the actual diagnosis ('M' = Malignant; 'B' = Benign).
- There are 30 other numeric features available for prediction.

Create Features and Labels

Split the data into 80% training, 10% validation and 10% testing.

In [5]:
```
rand_split = np.random.rand(len(data)) #Generates random numbers uniformly distributed in the range [0 and 1) ar
train_list = rand_split < 0.8
val_list = (rand_split >= 0.8) & (rand_split < 0.9)
test_list = rand_split >= 0.9

data_train = data[train_list]
data_val = data[val_list]
data_test = data[test_list]

train_y = ((data_train.iloc[:, 1] == "M") + 0).to_numpy() # you want all rows (:) of the second column (1). Cast
train_X = data_train.iloc[:, 2:].to_numpy()

val_y = ((data_val.iloc[:, 1] == "M") + 0).to_numpy()
val_X = data_val.iloc[:, 2:].to_numpy()

test_y = ((data_test.iloc[:, 1] == "M") + 0).to_numpy()
test_X = data_test.iloc[:, 2:].to_numpy();
```

Now, we'll convert the datasets to the recordIO-wrapped protobuf format (RecordIO file format and the Protocol Buffers (protobuf) serialization protocol, both of which are designed to efficiently handle large volumes of data) used by the Amazon SageMaker algorithms, and then upload this data to S3. See this page for the data format that this algorithm supports: https://docs.aws.amazon.com/sagemaker/latest/dg/common-info-all-im-models.html

We'll start with training data.

In [6]:
```
train_file = "linear_train.data" #sets the name of the file that will be created and eventually uploaded to Amaz

f = io.BytesIO() #creates an in-memory bytes buffer object. This buffer (f) object can be used as a file object

# In the following line, we write the training data to the BytesIO buffer in a dense tensor format.
# The train_X and train_y data are cast to the float32 type,
smac.write_numpy_to_dense_tensor(f, train_X.astype("float32"), train_y.astype("float32"))
f.seek(0)  #reset the file's position to the beginning.

boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "train", train_file)
).upload_fileobj(f)
```

Next we'll convert and upload the validation dataset.

In [7]:
```
validation_file = "linear_validation.data"

f = io.BytesIO()
smac.write_numpy_to_dense_tensor(f, val_X.astype("float32"), val_y.astype("float32"))
f.seek(0)

boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "validation", validation_file)
).upload_fileobj(f)
```

Check S3 bucket to see if they have been uploaded to S3.

Train

Now we can begin to specify our linear model. Amazon SageMaker's Linear Learner actually fits many models in parallel, each with slightly different hyperparameters, and then returns the one with the best fit. This functionality is automatically enabled. We can influence this using parameters like:

- `num_models` to increase to total number of models run. You should not confuse this with hyperparameter tuning that you will learn later. In this case the linear learner algorithm itself decides which hyperparameter values "nearby" the ones you provided to try out. You don't get to specify the "range or values" to explore, like the way you do in hyper parameter

tuning. The specified parameters will always based on what you set, but the algorithm also chooses models with nearby parameter values in order to find a solution nearby that may be more optimal. In this case, we're going to use the max of 32.

- `loss` which controls how we penalize mistakes in our model estimates. For this case, let's use absolute loss as we haven't spent much time cleaning the data, and absolute loss will be less sensitive to outliers.
- `wd` or `l1` which control regularization. Regularization can prevent model overfitting by preventing our estimates from becoming too finely tuned to the training data, which can actually hurt generalizability. In this case, we'll leave these parameters as their default "auto" though.

Specify container images used for training and hosting SageMaker's linear-learner

In [8]:
```
from sagemaker import image_uris

container = image_uris.retrieve(region=boto3.Session().region_name, framework="linear-learner")
```

In [9]:
```
linear_job = "DEMO-linear-" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())

print("Job name is:", linear_job)

linear_training_params = {
    "RoleArn": role,
    "TrainingJobName": linear_job,
    "AlgorithmSpecification": {"TrainingImage": container, "TrainingInputMode": "File"},
    "ResourceConfig": {"InstanceCount": 1, "InstanceType": "ml.c4.2xlarge", "VolumeSizeInGB": 10},
    "InputDataConfig": [
        {
            "ChannelName": "train",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": "s3://{}/{}/train/".format(bucket, prefix),
                    "S3DataDistributionType": "ShardedByS3Key",
                }
            },
            "CompressionType": "None",
            "RecordWrapperType": "None",
        },
        {
            "ChannelName": "validation",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": "s3://{}/{}/validation/".format(bucket, prefix),
                    "S3DataDistributionType": "FullyReplicated",
                }
            },
            "CompressionType": "None",
            "RecordWrapperType": "None",
        },
    ],
    "OutputDataConfig": {"S3OutputPath": "s3://{}/{}/".format(bucket, prefix)},
    "HyperParameters": {
        "feature_dim": "30",
        "mini_batch_size": "100",
        "predictor_type": "regressor",
        "epochs": "10",
        "num_models": "32",
        "loss": "absolute_loss",
    },
    "StoppingCondition": {"MaxRuntimeInSeconds": 60 * 60},
}
```

```
Job name is: DEMO-linear-2023-11-07-00-39-31
```

Now let's kick off our training job in SageMaker's distributed, managed training, using the parameters we just created. Because training is managed, we don't have to wait for our job to finish to continue, but for this case, let's use boto3's 'training_job_completed_or_stopped' waiter so we can ensure that the job has been started.

In [10]:
```
%%time

region = boto3.Session().region_name
sm = boto3.client("sagemaker")

sm.create_training_job(**linear_training_params)

status = sm.describe_training_job(TrainingJobName=linear_job)["TrainingJobStatus"]
print(status)
sm.get_waiter("training_job_completed_or_stopped").wait(TrainingJobName=linear_job)
if status == "Failed":
    message = sm.describe_training_job(TrainingJobName=linear_job)["FailureReason"]
    print("Training failed with the following error: {}".format(message))
    raise Exception("Training job failed")
```

```
InProgress
CPU times: user 127 ms, sys: 16.5 ms, total: 144 ms
Wall time: 4min
```

Host

Now that we've trained the linear algorithm on our data, let's setup a model which can later be hosted. We will:

1. Point to the scoring container
2. Point to the model.tar.gz that came from training
3. Create the hosting model

In [11]:
```
linear_hosting_container = {
    "Image": container,
    "ModelDataUrl": sm.describe_training_job(TrainingJobName=linear_job)["ModelArtifacts"][
        "S3ModelArtifacts"
    ],
}

create_model_response = sm.create_model(
    ModelName=linear_job, ExecutionRoleArn=role, PrimaryContainer=linear_hosting_container
)

print(create_model_response["ModelArn"])
```

```
arn:aws:sagemaker:us-east-1:239630988601:model/demo-linear-2023-11-07-00-39-31
```

Once we've setup a model, we can configure what our hosting endpoints should be. Here we specify:

1. EC2 instance type to use for hosting
2. Initial number of instances
3. Our hosting model name

In [12]:
```
linear_endpoint_config = "DEMO-linear-endpoint-config-" + time.strftime(
    "%Y-%m-%d-%H-%M-%S", time.gmtime()
)
print(linear_endpoint_config)
create_endpoint_config_response = sm.create_endpoint_config(
    EndpointConfigName=linear_endpoint_config,
    ProductionVariants=[
        {
            "InstanceType": "ml.m4.xlarge",
            "InitialInstanceCount": 1,
            "ModelName": linear_job,
            "VariantName": "AllTraffic",
        }
    ],
)

print("Endpoint Config Arn: " + create_endpoint_config_response["EndpointConfigArn"])
```

```
DEMO-linear-endpoint-config-2023-11-07-01-37-39
Endpoint Config Arn: arn:aws:sagemaker:us-east-1:239630988601:endpoint-config/demo-linear-endpoint-config-2023-11-07-01-37-39
```

Now that we've specified how our endpoint should be configured, we can create them. This can be done in the background, but for now let's run a loop that updates us on the status of the endpoints so that we know when they are ready for use.

In [13]:
```
%%time

linear_endpoint = "DEMO-linear-endpoint-" + time.strftime("%Y%m%d%H%M", time.gmtime())
print(linear_endpoint)
create_endpoint_response = sm.create_endpoint(
    EndpointName=linear_endpoint, EndpointConfigName=linear_endpoint_config
)
print(create_endpoint_response["EndpointArn"])

resp = sm.describe_endpoint(EndpointName=linear_endpoint)
status = resp["EndpointStatus"]
print("Status: " + status)

sm.get_waiter("endpoint_in_service").wait(EndpointName=linear_endpoint)

resp = sm.describe_endpoint(EndpointName=linear_endpoint)
status = resp["EndpointStatus"]
print("Arn: " + resp["EndpointArn"])
print("Status: " + status)

if status != "InService":
    raise Exception("Endpoint creation did not succeed")
```

```
DEMO-linear-endpoint-202311070138
arn:aws:sagemaker:us-east-1:239630988601:endpoint/demo-linear-endpoint-202311070138
Status: Creating
Arn: arn:aws:sagemaker:us-east-1:239630988601:endpoint/demo-linear-endpoint-202311070138
Status: InService
CPU times: user 82.4 ms, sys: 12 ms, total: 94.3 ms
Wall time: 4min 1s
```

Predict

Predict on Test Data

Now that we have our hosted endpoint, we can generate statistical predictions from it. Let's predict on our test dataset to understand how accurate our model is.

There are many metrics to measure classification accuracy. Common examples include include:

- Precision
- Recall
- F1 measure
- Area under the ROC curve - AUC
- Total Classification Accuracy
- Mean Absolute Error

For our example, we'll keep things simple and use total classification accuracy as our metric of choice. We will also evaluate Mean Absolute Error (MAE) as the linear-learner has been optimized using this metric, not necessarily because it is a relevant metric from an application point of view. We'll compare the performance of the linear-learner against a naive benchmark prediction which uses majority class observed in the training data set for prediction on the test data.

Function to convert an array to a csv

In [14]:
```
def np2csv(arr):
    csv = io.BytesIO() #the function gets an array (Numpy array) and creates an in-memory binary buffer named cs
    np.savetxt(csv, arr, delimiter=",", fmt="%g") # write the array 'arr' to csv object, columns should be seper
    # In the following line:
```

```
    # csv.getvalue() retrieves the entire contents of the buffer csv as a byte string.
    # .decode() converts the byte string into a normal Python string by decoding it using the default UTF-8 enco
    #.rstrip() removes any trailing whitespace or newlines from the end of the string.
    return csv.getvalue().decode().rstrip()
```

Next, we'll invoke the endpoint to get predictions.

In [15]:
```
runtime = boto3.client("runtime.sagemaker")

payload = np2csv(test_X)

response = runtime.invoke_endpoint(
    EndpointName=linear_endpoint, ContentType="text/csv", Body=payload
)
result = json.loads(response["Body"].read().decode())
#The following line:
# extracts the prediction results from the result dictionary, from the "predictions" key.
# It is a list of dictionaries where each dictionary has a key "score" representing the model's prediction.
#A list comprehension is used to create a list of these scores, which is then converted into a NumPy array calle
test_pred = np.array([r["score"] for r in result["predictions"]])
```

In [16]:
```
print (test_pred)
```

```
[ 1.03552222e+00  4.07223284e-01  8.14391434e-01  5.18647373e-01
  9.91151392e-01  1.42991006e-01  5.45164049e-01 -1.46963656e-01
 -2.93104589e-01  7.47275949e-02  7.93953538e-02  8.16240370e-01
 -6.65979981e-02  2.92676985e-01 -1.29966140e-02  1.23876154e-01
  8.00839484e-01 -9.50319767e-02 -7.57524371e-02 -1.67738616e-01
  8.80197883e-02  1.19211149e+00 -1.99126780e-01  9.27334428e-02
  2.46621430e-01  1.09628725e+00  3.54275107e-02  7.29543149e-01
  2.52020419e-01  6.94946826e-01  7.62482464e-01  7.88641036e-01
  1.14349723e-02  2.57428110e-01  3.74307454e-01  9.43569601e-01
  9.25443769e-02  5.64817488e-01 -1.50046468e-01  3.70727777e-02
 -7.26755261e-02  9.10505354e-01  1.42713487e-01 -1.07467353e-01
  2.25648284e-02 -1.01560950e-02  1.17336297e+00 -1.46628022e-02
 -3.23155701e-01 -1.43141508e-01  1.38597071e-01  2.13433683e-01
 -1.97946429e-02  9.51290131e-05  1.26083207e+00]
```

Let's compare linear learner based mean absolute prediction errors from a baseline prediction which uses majority class to predict every instance.

In [17]:
```
test_mae_linear = np.mean(np.abs(test_y - test_pred)) # Mean Absolute Error (MAE) of the predictions vs real tar
#The following line calculates the MAE baseline model using a very simple strategy for predictions:
#it always predicts the median value of the target variable from the training dataset, which is np.median(train_
#The idea is to provide a simple comparison to see if the linear model is performing better than a model that al
test_mae_baseline = np.mean(
    np.abs(test_y - np.median(train_y))
)

print("Test MAE Baseline :", round(test_mae_baseline, 3))
print("Test MAE Linear:", round(test_mae_linear, 3))
```

```
Test MAE Baseline : 0.34500000000000003
Test MAE Linear: 0.165
```

Let's undrestand the above values:

Baseline Model (MAE 0.353): This model, which always predicts the median value of the training target variable, has an average absolute error of 0.353. This gives you a reference point for the minimum performance any reasonable model should beat, since it's a very simple approach that doesn't use any features from the input data.

Linear Model (MAE 0.201): Your linear model has an average absolute error of 0.201, which is substantially lower than the baseline's MAE. This improvement indicates that the linear model is successfully leveraging the features in the input data to make more accurate predictions than the simple strategy of always guessing the median.

In practical terms, the linear model is reducing the error in predictions by about 43% compared to the baseline model (calculated as (0.353 - 0.201) / 0.353). This is a substantial improvement and suggests that the linear model has learned a meaningful representation of the relationship between the input features and the target variable.

Let's compare predictive accuracy using a classification threshold of 0.5 for the predicted and compare against the majority class prediction from training data set.

In [18]:
```
#If the predicted probability (test_pred) is greater than 0.5, it is considered a prediction for the positive cl
#The + 0 part is used to convert the resulting boolean values (True for values above 0.5 and False for values be
#into integers (1 and 0, respectively)
test_pred_class = (test_pred > 0.5) + 0

# We sort the train_y values and select the mdedian (if is 0, then we select 0 otherwise 1). Then we generate th
#to the lenght of test_y. We use this as baseline for test_pred_baseline
test_pred_baseline = np.repeat(np.median(train_y), len(test_y))

#compare the binary prediction (test_pred_class) with actual outcomes (test_y)
prediction_accuracy = np.mean((test_y == test_pred_class)) * 100
baseline_accuracy = np.mean((test_y == test_pred_baseline)) * 100

print("Prediction Accuracy:", round(prediction_accuracy, 1), "%")
print("Baseline Accuracy:", round(baseline_accuracy, 1), "%")
```

```
Prediction Accuracy: 98.2 %
Baseline Accuracy: 65.5 %
```

Again, let's undestand these values:

Prediction Accuracy: 96.1% This value represents the percentage of the test dataset that your model predicted correctly. In other words, 96.1% of the time, the model's prediction matched the actual target value. An accuracy of over 96% suggests that your model is performing very well on the test data, assuming the test set is representative of the real-world data the model will encounter.

Baseline Accuracy: 64.7% This figure is the accuracy of the baseline model, which always predicts the median value of the target variable from the training dataset. If your target variable is binary, the median will often be the most common class in a balanced or nearly balanced dataset. Therefore, an accuracy of 64.7% suggests that by always predicting the median value of the training targets, you would be correct 64.7% of the time.

Cleanup

In [19]:
```
sm.delete_endpoint(EndpointName=linear_endpoint)
```

Out[19]:
```
{'ResponseMetadata': {'RequestId': 'c2f83145-da10-4cf0-8b79-c6b01eb0c511',
  'HTTPStatusCode': 200,
  'HTTPHeaders': {'x-amzn-requestid': 'c2f83145-da10-4cf0-8b79-c6b01eb0c511',
   'content-type': 'application/x-amz-json-1.1',
   'content-length': '0',
   'date': 'Tue, 07 Nov 2023 01:59:54 GMT'},
  'RetryAttempts': 0}}
```

Question

- Our linear model does a good job of predicting breast cancer and has an overall accuracy of close to 90%+. We want to see how that would be different if we were using a different model that Linear Learner, models such as XGBoost or other models. You can also use differenr hyperparameters.
- We also did not do much feature engineering. We could create additional features by considering cross-product/interaction of multiple features, squaring or raising higher powers of the features to induce non-linear effects, etc. If we expand the features using non-linear terms and interactions, we can then tweak the regulaization parameter to optimize the expanded model and hence generate improved forecasts.
- Try some of the above techniques and see what changes when it comes to Prediction Accuracy and Baseline Accuracy as I have expalined above.

In [123…
```
import os
import boto3
import re
import sagemaker

role = sagemaker.get_execution_role()
region = boto3.Session().region_name

# S3 bucket for saving code and model artifacts.
# Feel free to specify a different bucket and prefix
bucket = "day07-mk1-realtime-serverless"

prefix = (
    "sagemaker/DEMO-breast-cancer-prediction"  # place to upload training files within the bucket
)
print (region)
```

```
sagemaker.config INFO - Not applying SDK defaults from location: /etc/xdg/sagemaker/config.yaml
sagemaker.config INFO - Not applying SDK defaults from location: /home/ec2-user/.config/sagemaker/config.yaml
us-east-1
```

In [124…
```
role
```

Out[124]:
```
'arn:aws:iam::585522057818:role/fast-ai-academic-11-Student-Azure'
```

In [125…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import io
import time
import json
import sagemaker.amazon.common as smac
```

In [127…
```
s3 = boto3.client("s3")

filename = "wdbc.csv"
s3.download_file("sagemaker-sample-files", "datasets/tabular/breast_cancer/wdbc.csv", filename)
data = pd.read_csv(filename, header=None)

# specify columns extracted from wbdc.names
data.columns = [
    "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",
]

# save the data
data.to_csv("data.csv", sep=",", index=False)

# print the shape of the data file
print(data.shape)

# show the top few rows
display(data.head())

# describe the data object
display(data.describe())

# we will also summarize the categorical field diganosis
display(data.diagnosis.value_counts())
```

```
(569, 32)
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_m
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.

5 rows × 32 columns

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_me
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.0000
mean	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.0887
std	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.0797
min	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.0000
25%	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.0295
50%	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.0615
75%	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.1307
max	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.4268

8 rows × 31 columns

```
diagnosis
B    357
M    212
Name: count, dtype: int64
```

In [164…

```
rand_split = np.random.rand(len(data)) #Generates random numbers uniformly distributed in the range [0 and 1) ar
train_list = rand_split < 0.8
val_list = (rand_split >= 0.8) & (rand_split < 0.9)
test_list = rand_split >= 0.9

data_train = data[train_list]
data_val = data[val_list]
data_test = data[test_list]

train_y = ((data_train.iloc[:, 1] == "M") + 0).to_numpy() # you want all rows (:) of the second column (1). Cas
train_X = data_train.iloc[:, 2:].to_numpy()

val_y = ((data_val.iloc[:, 1] == "M") + 0).to_numpy()
val_X = data_val.iloc[:, 2:].to_numpy()
```

```
test_y = ((data_test.iloc[:, 1] == "M") + 0).to_numpy()
test_X = data_test.iloc[:, 2:].to_numpy();
```

In [165…
```
# train_file = "linear_train.data" #sets the name of the file that will be created and eventually uploaded to Am

# f = io.BytesIO() #creates an in-memory bytes buffer object. This buffer (f) object can be used as a file objec

# # In the following line, we write the training data to the BytesIO buffer in a dense tensor format.
# # The train_X and train_y data are cast to the float32 type,
# smac.write_numpy_to_dense_tensor(f, train_X.astype("float32"), train_y.astype("float32"))
# f.seek(0)  #reset the file's position to the beginning.

# boto3.Session().resource("s3").Bucket(bucket).Object(
#     os.path.join(prefix, "train", train_file)
# ).upload_fileobj(f)

train_file = "xgboost_train.csv" #sets the name of the file that will be created and eventually uploaded to Amaz

# Convert and save training set to CSV
train_df = pd.DataFrame(np.column_stack((train_y, train_X)))

train_df.to_csv(train_file, index=False, header=False)

boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "xg_train", train_file)
).upload_file(train_file)
```

In [166…
```
# validation_file = "linear_validation.data"

# f = io.BytesIO()
# smac.write_numpy_to_dense_tensor(f, val_X.astype("float32"), val_y.astype("float32"))
# f.seek(0)

# boto3.Session().resource("s3").Bucket(bucket).Object(
#     os.path.join(prefix, "validation", validation_file)
# ).upload_fileobj(f)

validation_file = "xgboost_validation.csv"
val_df = pd.DataFrame(np.column_stack((val_y, val_X)))

val_df.to_csv(validation_file, index=False, header=False)

boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "xg_validation", validation_file)
).upload_file(validation_file)
```

Train - xgboost

In [167…
```
from sagemaker import image_uris

container = image_uris.retrieve(region=boto3.Session().region_name, framework="xgboost", version='1.5-1')
```

In [ ]:

In [ ]:

In [168…
```
xgboost_job = "DEMO-xgboost-" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())
print("Job name is:", xgboost_job)

s3_input_train = "s3://{}/{}/xg_train".format(bucket, prefix)
s3_input_validation = "s3://{}/{}/xg_validation/".format(bucket, prefix)

xgboost_training_params = {
    "RoleArn": role,
    "TrainingJobName": xgboost_job,
    "AlgorithmSpecification": {"TrainingImage": container, "TrainingInputMode": "File"},
    "ResourceConfig": {"InstanceCount": 1, "InstanceType": "ml.m4.xlarge", "VolumeSizeInGB": 10},
    "InputDataConfig": [
```

```
        {
            "ChannelName": "train",
            "CompressionType": "None",
            "ContentType": "csv",
            "RecordWrapperType": "None",
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3_input_train,
                }
            },
        },
        {
            "ChannelName": "validation",
            "CompressionType": "None",
            "ContentType": "csv",
            "RecordWrapperType": "None",
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3_input_validation,
                }
            },
        },
    ],
    "OutputDataConfig": {"S3OutputPath": "s3://{}/{}/output".format(bucket, prefix)},
    "HyperParameters": {
        "eval_metric": "mae",
        "num_round": "100",
        "objective": "binary:logistic",
        "rate_drop": "0.3",
    },
    "StoppingCondition": {"MaxRuntimeInSeconds": 3600},
}
```

```
Job name is: DEMO-xgboost-2023-11-14-07-54-41
```

In [169…

```
%%time

region = boto3.Session().region_name
sm = boto3.client("sagemaker")

sm.create_training_job(**xgboost_training_params)

status = sm.describe_training_job(TrainingJobName=xgboost_job)["TrainingJobStatus"]
print(status)
sm.get_waiter("training_job_completed_or_stopped").wait(TrainingJobName=xgboost_job)
if status == "Failed":
    message = sm.describe_training_job(TrainingJobName=xgboost_job)["FailureReason"]
    print("Training failed with the following error: {}".format(message))
    raise Exception("Training job failed")
```

```
InProgress
CPU times: user 117 ms, sys: 3.68 ms, total: 121 ms
Wall time: 4min
```

In [ ]:

```
```

Hosting - xgboost

In [170…

```
xgboost_hosting_container = {
    "Image": container,
    "ModelDataUrl": sm.describe_training_job(TrainingJobName=xgboost_job)["ModelArtifacts"][
        "S3ModelArtifacts"
    ],
}

create_model_response = sm.create_model(
    ModelName=xgboost_job, ExecutionRoleArn=role, PrimaryContainer=xgboost_hosting_container
```

```
)

print(create_model_response["ModelArn"])
```

```
arn:aws:sagemaker:us-east-1:585522057818:model/demo-xgboost-2023-11-14-07-54-41
```

In [171…

```
xgboost_endpoint_config = "DEMO-xgboost-endpoint-config-" + time.strftime(
    "%Y-%m-%d-%H-%M-%S", time.gmtime()
)
print(xgboost_endpoint_config)
create_endpoint_config_response = sm.create_endpoint_config(
    EndpointConfigName=xgboost_endpoint_config,
    ProductionVariants=[
        {
            "InstanceType": "ml.m4.xlarge",
            "InitialInstanceCount": 1,
            "ModelName": xgboost_job,
            "VariantName": "AllTraffic",
        }
    ],
)

print("Endpoint Config Arn: " + create_endpoint_config_response["EndpointConfigArn"])
```

```
DEMO-xgboost-endpoint-config-2023-11-14-07-59-38
Endpoint Config Arn: arn:aws:sagemaker:us-east-1:585522057818:endpoint-config/demo-xgboost-endpoint-config-2023
-11-14-07-59-38
```

In [172…

```
%%time

xgboost_endpoint = "DEMO-xgboost-endpoint-" + time.strftime("%Y%m%d%H%M", time.gmtime())
print(xgboost_endpoint)
create_endpoint_response = sm.create_endpoint(
    EndpointName=xgboost_endpoint, EndpointConfigName=xgboost_endpoint_config
)
print(create_endpoint_response["EndpointArn"])

resp = sm.describe_endpoint(EndpointName=xgboost_endpoint)
status = resp["EndpointStatus"]
print("Status: " + status)

sm.get_waiter("endpoint_in_service").wait(EndpointName=xgboost_endpoint)

resp = sm.describe_endpoint(EndpointName=xgboost_endpoint)
status = resp["EndpointStatus"]
print("Arn: " + resp["EndpointArn"])
print("Status: " + status)

if status != "InService":
    raise Exception("Endpoint creation did not succeed")
```

```
DEMO-xgboost-endpoint-202311140759
arn:aws:sagemaker:us-east-1:585522057818:endpoint/demo-xgboost-endpoint-202311140759
Status: Creating
Arn: arn:aws:sagemaker:us-east-1:585522057818:endpoint/demo-xgboost-endpoint-202311140759
Status: InService
CPU times: user 70.6 ms, sys: 4.5 ms, total: 75.1 ms
Wall time: 4min 31s
```

Predict - xgboost

In [173…

```
def np2csv(arr):
    csv = io.BytesIO() #the function gets an array (Numpy array) and creates an in-memory binary buffer named cs
    np.savetxt(csv, arr, delimiter=",", fmt="%g") # write the array 'arr' to csv object, columns should be seper
    # In the following line:
    # csv.getvalue() retrieves the entire contents of the buffer csv as a byte string.
    # .decode() converts the byte string into a normal Python string by decoding it using the default UTF-8 enco
    #.rstrip() removes any trailing whitespace or newlines from the end of the string.
    return csv.getvalue().decode().rstrip()
```

In [174…

```
%%time
```

```
runtime = boto3.client("runtime.sagemaker")

payload = np2csv(test_X)

response = runtime.invoke_endpoint(
    EndpointName=xgboost_endpoint, ContentType="text/csv", Body=payload
)
result = response["Body"].read().decode()
test_pred = np.array(result.split(), dtype=float)
```

```
CPU times: user 28.2 ms, sys: 0 ns, total: 28.2 ms
Wall time: 178 ms
```

In [175…
```
print(test_pred)
```

```
[9.95462120e-01 9.99238014e-01 9.99564826e-01 3.37421715e-01
 9.96182263e-01 9.99253452e-01 9.55933283e-05 1.40619678e-02
 2.05547595e-03 9.99810517e-01 4.29976091e-04 2.07405363e-04
 7.40944803e-01 2.63892680e-01 5.30597288e-03 6.26707776e-03
 7.85229553e-04 9.99627113e-01 2.71959289e-04 9.99731362e-01
 9.58046615e-01 1.68183193e-04 9.99527574e-01 9.99603689e-01
 9.99730647e-01 9.98016715e-01 9.99932289e-01 9.85065162e-01
 7.75096589e-04 6.18446469e-02 5.94769546e-04 9.99816120e-01
 6.23460219e-04 9.99775589e-01 9.99141216e-01 7.72710555e-05
 5.56180603e-04 9.99648213e-01 5.00403577e-04 1.17164993e-04
 1.03421346e-03 1.34853821e-03 2.52287020e-04 9.99786317e-01
 7.79677008e-04 9.98223364e-01 9.99750435e-01 2.08110563e-04
 8.87577515e-03 1.84460566e-03 1.43403784e-04 5.66925970e-04
 3.28541920e-02 9.99752581e-01]
```

In [176…
```
test_mae_linear = np.mean(np.abs(test_y - test_pred)) # Mean Absolute Error (MAE) of the predictions vs real tar
#The following line calculates the MAE baseline model using a very simple strategy for predictions:
#it always predicts the median value of the target variable from the training dataset, which is np.median(train_
#The idea is to provide a simple comparison to see if the linear model is performing better than a model that al
test_mae_baseline = np.mean(
    np.abs(test_y - np.median(train_y))
)

print("Test MAE Baseline :", round(test_mae_baseline, 3))
print("Test MAE Linear:", round(test_mae_linear, 3))
```

```
Test MAE Baseline : 0.463
Test MAE Linear: 0.026
```

In [177…
```
#If the predicted probability (test_pred) is greater than 0.5, it is considered a prediction for the positive cl
#The + 0 part is used to convert the resulting boolean values (True for values above 0.5 and False for values be
#into integers (1 and 0, respectively)
test_pred_class = (test_pred > 0.5) + 0

# We sort the train_y values and select the mdedian (if is 0, then we select 0 otherwise 1). Then we generate th
#to the lenght of test_y. We use this as baseline for test_pred_baseline
test_pred_baseline = np.repeat(np.median(train_y), len(test_y))

#compare the binary prediction (test_pred_class) with actual outcomes (test_y)
prediction_accuracy = np.mean((test_y == test_pred_class)) * 100
baseline_accuracy = np.mean((test_y == test_pred_baseline)) * 100

print("Prediction Accuracy:", round(prediction_accuracy, 1), "%")
print("Baseline Accuracy:", round(baseline_accuracy, 1), "%")
```

```
Prediction Accuracy: 98.1 %
Baseline Accuracy: 53.7 %
```

In [178…
```
sm.delete_endpoint(EndpointName=xgboost_endpoint)
```

Out[178]:
```
{'ResponseMetadata': {'RequestId': '507d9560-dedc-4d28-8a73-c6320e96d2e5',
  'HTTPStatusCode': 200,
  'HTTPHeaders': {'x-amzn-requestid': '507d9560-dedc-4d28-8a73-c6320e96d2e5',
   'content-type': 'application/x-amz-json-1.1',
   'content-length': '0',
   'date': 'Tue, 14 Nov 2023 08:04:38 GMT'},
  'RetryAttempts': 0}}
```

In [ ]:

Train - Linear

In [45]:
```
from sagemaker import image_uris

container = image_uris.retrieve(region=boto3.Session().region_name, framework="linear-learner")
```

In [46]:
```
linear_job = "DEMO-linear-" + time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())

print("Job name is:", linear_job)

linear_training_params = {
    "RoleArn": role,
    "TrainingJobName": linear_job,
    "AlgorithmSpecification": {"TrainingImage": container, "TrainingInputMode": "File"},
    "ResourceConfig": {"InstanceCount": 1, "InstanceType": "ml.c4.2xlarge", "VolumeSizeInGB": 10},
    "InputDataConfig": [
        {
            "ChannelName": "train",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": "s3://{}/{}/train/".format(bucket, prefix),
                    "S3DataDistributionType": "ShardedByS3Key",
                }
            },
            "CompressionType": "None",
            "RecordWrapperType": "None",
        },
        {
            "ChannelName": "validation",
            "DataSource": {
                "S3DataSource": {
                    "S3DataType": "S3Prefix",
                    "S3Uri": "s3://{}/{}/validation/".format(bucket, prefix),
                    "S3DataDistributionType": "FullyReplicated",
                }
            },
            "CompressionType": "None",
            "RecordWrapperType": "None",
        },
    ],
    "OutputDataConfig": {"S3OutputPath": "s3://{}/{}/".format(bucket, prefix)},
    "HyperParameters": {
        "feature_dim": "30",
        "mini_batch_size": "100",
        "predictor_type": "regressor",
        "epochs": "10",
        "num_models": "32",
        "loss": "absolute_loss",
    },
    "StoppingCondition": {"MaxRuntimeInSeconds": 60 * 60},
}
```

```
Job name is: DEMO-linear-2023-11-13-23-33-41
```

In [47]:
```
%%time

region = boto3.Session().region_name
sm = boto3.client("sagemaker")

sm.create_training_job(**linear_training_params)

status = sm.describe_training_job(TrainingJobName=linear_job)["TrainingJobStatus"]
print(status)
sm.get_waiter("training_job_completed_or_stopped").wait(TrainingJobName=linear_job)
if status == "Failed":
    message = sm.describe_training_job(TrainingJobName=linear_job)["FailureReason"]
```

```
    print("Training failed with the following error: {}".format(message))
    raise Exception("Training job failed")
```

```
InProgress
CPU times: user 138 ms, sys: 5.04 ms, total: 143 ms
Wall time: 6min
```

In [48]:
```
linear_hosting_container = {
    "Image": container,
    "ModelDataUrl": sm.describe_training_job(TrainingJobName=linear_job)["ModelArtifacts"][
        "S3ModelArtifacts"
    ],
}

create_model_response = sm.create_model(
    ModelName=linear_job, ExecutionRoleArn=role, PrimaryContainer=linear_hosting_container
)

print(create_model_response["ModelArn"])
```

```
arn:aws:sagemaker:us-east-1:585522057818:model/demo-linear-2023-11-13-23-33-41
```

In [49]:
```
linear_endpoint_config = "DEMO-linear-endpoint-config-" + time.strftime(
    "%Y-%m-%d-%H-%M-%S", time.gmtime()
)
print(linear_endpoint_config)
create_endpoint_config_response = sm.create_endpoint_config(
    EndpointConfigName=linear_endpoint_config,
    ProductionVariants=[
        {
            "InstanceType": "ml.m4.xlarge",
            "InitialInstanceCount": 1,
            "ModelName": linear_job,
            "VariantName": "AllTraffic",
        }
    ],
)

print("Endpoint Config Arn: " + create_endpoint_config_response["EndpointConfigArn"])
```

```
DEMO-linear-endpoint-config-2023-11-13-23-40-24
Endpoint Config Arn: arn:aws:sagemaker:us-east-1:585522057818:endpoint-config/demo-linear-endpoint-config-2023-
11-13-23-40-24
```

In [50]:
```
%%time

linear_endpoint = "DEMO-linear-endpoint-" + time.strftime("%Y%m%d%H%M", time.gmtime())
print(linear_endpoint)
create_endpoint_response = sm.create_endpoint(
    EndpointName=linear_endpoint, EndpointConfigName=linear_endpoint_config
)
print(create_endpoint_response["EndpointArn"])

resp = sm.describe_endpoint(EndpointName=linear_endpoint)
status = resp["EndpointStatus"]
print("Status: " + status)

sm.get_waiter("endpoint_in_service").wait(EndpointName=linear_endpoint)

resp = sm.describe_endpoint(EndpointName=linear_endpoint)
status = resp["EndpointStatus"]
print("Arn: " + resp["EndpointArn"])
print("Status: " + status)

if status != "InService":
    raise Exception("Endpoint creation did not succeed")
```

```
DEMO-linear-endpoint-202311132340
arn:aws:sagemaker:us-east-1:585522057818:endpoint/demo-linear-endpoint-202311132340
Status: Creating
Arn: arn:aws:sagemaker:us-east-1:585522057818:endpoint/demo-linear-endpoint-202311132340
Status: InService
CPU times: user 80.3 ms, sys: 6.76 ms, total: 87 ms
Wall time: 4min 31s
```

Predict-linear

In [51]:
```
def np2csv(arr):
    csv = io.BytesIO() #the function gets an array (Numpy array) and creates an in-memory binary buffer named cs
    np.savetxt(csv, arr, delimiter=",", fmt="%g") # write the array 'arr' to csv object, columns should be seper
    # In the following line:
    # csv.getvalue() retrieves the entire contents of the buffer csv as a byte string.
    # .decode() converts the byte string into a normal Python string by decoding it using the default UTF-8 enco
    #.rstrip() removes any trailing whitespace or newlines from the end of the string.
    return csv.getvalue().decode().rstrip()
```

In [52]:
```
%%time
runtime = boto3.client("runtime.sagemaker")

payload = np2csv(test_X)

response = runtime.invoke_endpoint(
    EndpointName=linear_endpoint, ContentType="text/csv", Body=payload
)
result = json.loads(response["Body"].read().decode())
#The following line:
# extracts the prediction results from the result dictionary, from the "predictions" key.
# It is a list of dictionaries where each dictionary has a key "score" representing the model's prediction.
#A list comprehension is used to create a list of these scores, which is then converted into a NumPy array calle
test_pred = np.array([r["score"] for r in result["predictions"]])
```

```
CPU times: user 25.8 ms, sys: 2.51 ms, total: 28.3 ms
Wall time: 210 ms
```

In [53]:
```
print(test_pred)
```

```
[ 1.65868640e+00  1.07613635e+00  1.36831641e+00  1.29709196e+00
  2.34603882e-03  8.92233610e-01  1.85680890e+00  3.24442387e-01
  1.64989710e-01  4.90528345e-02  1.94222856e+00  5.80971241e-01
  6.41139746e-02  8.49728107e-01 -2.64894485e-01  7.85136700e-01
  4.50833321e-01  9.59813595e-03  2.36891389e-01  4.85881567e-02
  1.71613693e-01  8.72616529e-01 -2.07571149e-01 -9.71388817e-03
  1.03277469e+00  8.80339146e-02  4.32660818e-01  1.38537812e+00
  2.90467739e-01  3.79951715e-01 -2.51617432e-01  2.15289116e-01
  6.86006069e-01  1.68536544e+00 -8.38136673e-02  2.41053224e-01
 -1.49030685e-02 -5.35283089e-02  4.53337669e-01  1.92597628e-01
  1.20253778e+00  1.11924076e+00  3.04261923e-01 -1.68099284e-01
  2.40102839e+00  2.02758431e-01 -1.15429163e-01  3.25849771e-01
  5.10718584e-01  5.07594347e-02  9.32420492e-02 -5.97903728e-02
 -6.60431385e-02 -6.09774590e-02  3.31310034e-01]
```

In [54]:
```
test_mae_linear = np.mean(np.abs(test_y - test_pred)) # Mean Absolute Error (MAE) of the predictions vs real tar
#The following line calculates the MAE baseline model using a very simple strategy for predictions:
#it always predicts the median value of the target variable from the training dataset, which is np.median(train_
#The idea is to provide a simple comparison to see if the linear model is performing better than a model that al
test_mae_baseline = np.mean(
    np.abs(test_y - np.median(train_y))
)

print("Test MAE Baseline :", round(test_mae_baseline, 3))
print("Test MAE Linear:", round(test_mae_linear, 3))
```

```
Test MAE Baseline : 0.4
Test MAE Linear: 0.266
```

In [55]:
```
#If the predicted probability (test_pred) is greater than 0.5, it is considered a prediction for the positive cl
#The + 0 part is used to convert the resulting boolean values (True for values above 0.5 and False for values be
#into integers (1 and 0, respectively)
test_pred_class = (test_pred > 0.5) + 0
```

```
# We sort the train_y values and select the mdedian (if is 0, then we select 0 otherwise 1). Then we generate th
#to the lenght of test_y. We use this as baseline for test_pred_baseline
test_pred_baseline = np.repeat(np.median(train_y), len(test_y))

#compare the binary prediction (test_pred_class) with actual outcomes (test_y)
prediction_accuracy = np.mean((test_y == test_pred_class)) * 100
baseline_accuracy = np.mean((test_y == test_pred_baseline)) * 100

print("Prediction Accuracy:", round(prediction_accuracy, 1), "%")
print("Baseline Accuracy:", round(baseline_accuracy, 1), "%")
```

```
Prediction Accuracy: 94.5 %
Baseline Accuracy: 60.0 %
```

cleanup

In [56]:
```
sm.delete_endpoint(EndpointName=linear_endpoint)
```

Out[56]:
```
{'ResponseMetadata': {'RequestId': 'b4aa3ede-bbda-43e7-b3e1-000ac30d7880',
  'HTTPStatusCode': 200,
  'HTTPHeaders': {'x-amzn-requestid': 'b4aa3ede-bbda-43e7-b3e1-000ac30d7880',
   'content-type': 'application/x-amz-json-1.1',
   'content-length': '0',
   'date': 'Mon, 13 Nov 2023 23:52:09 GMT'},
  'RetryAttempts': 0}}
```

In [ ]:
```

```