Bird Sound Recognition Web-App

Alexandru Melnic (1692625), Filippo Mastrogiacomo (1891292),
Dario Russo (1714011), Dilara Isikli (1891292)

September 6, 2021

1 Introduction

We built a Web-App using AWS technologies. The aim of the Web-App is to classify bird species from their sound. The user can upload its sound on a static and serverless Web-App, which exploits AWS technologies to process (Lambda) and classify (SageMaker) it. The web site is available HERE and clicking on this LINK it is possible to get the access to the repository of this project.

2 The Architecture

The idea is to develop a serverless application that thanks to its ability to scale can deal with an important amount of requests per second. To achieve this goal, we have exploited some of the most known AWS supplied services, such as Lambda and SageMaker. The Web-App is hosted on a public S3 bucket. The interaction between the Web-App and the back-end is made possible by the API Gateway, through which is possible to call the main Lambda The Lambda then sends a request to SageMaker that returns as response the bird to which corresponds that sound with the highest probability.

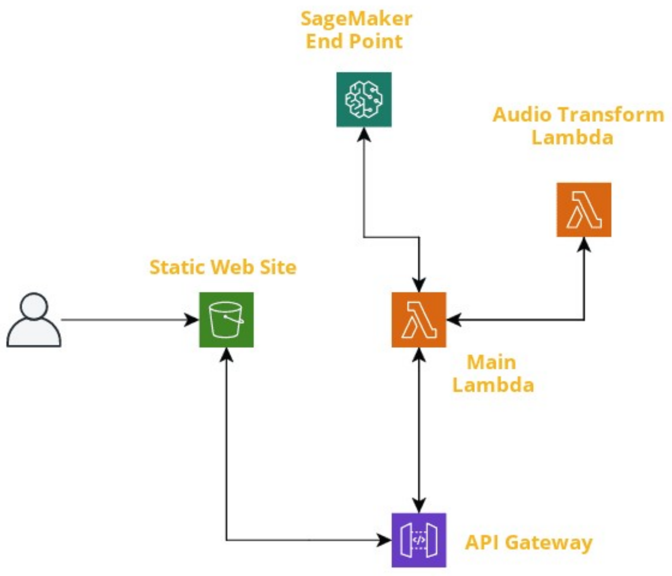

Figure 1: The Architecture of the Web App

2.1 Web App Hosting

The Web-App is hosted on an a public S3 bucket, as the name says we added a policy to make any file in it automatically public. Then, we set the bucket's property so that when the bucket's endpoint is called, the index.html file is invoked.

Another possible Web-Approach would have been to host the Web-App on dedicate serves, using AWS EC2, which would have been used to serve the static files and the application logic. However, our choice to host the Web-App on an S3 presents at least 4 points of strength, which make it far away more convenient than the hosting on the EC2s:

- **Cheaper:** on the S3 the user is billed only for the space that he/she uses. Hence, there is no need to manage disk utilisation.
- **Easy Use and Maintenance:** with our approach we just have to upload the files on the S3, without the need to use and manage any server.
- **Scalability:** when more resources are needed the storage is automatically increased, without any need of configuration like in a server.
- **Performance Improved:** Static resources are always loaded faster.

In addition, to ensure a low latency and higher performance, it is possible to configure the Cloud-Front distribution. We decided to not implement it, because our Web-App is thought to be delivered only in Europe. Therefore, to reduce costs, at least in a first moment, when the Web-App is used only by few users, we won't use this service.

2.2 Lambda Functions

AWS Lambda is a compute service that let the user run code without provisioning or managing servers. It is great solution, if you want a scalable service with high performance. Our architecture contains two Lambda, for two reason:

- **Optimization:** splitting complex processes into separate functions help to save money and gain speed (especially when functions can be run in an asynchronous way).
- **Layers Limitation:** Lambda has installed only a few libraries, anyway it is possible to provide the needed libraries using the layers. However, the size of all libraries contained in different layers of the same Lambda cannot exceed 250 MB uncompressed.

Our main Lambda works with SageMaker, so it is important to configure it with the correct IAM roles, which allow the access to the end point. To be sure that everything works well we give full permission to the Lambda, but to be safer, it would be better to give only the permission strictly needed. Then we give to both the Lambdas the access to CloudWatch to have better insights.

On the first Lambda, which can be called the main one, we extract from the packet sent from the website the audio encoded in the Base64 format and we convert again to an audio file by using `pydub` and `ffmpeg`, which are installed in a layer of the Lambda. Then the audio is sent to the second Lambda, which transforms it in the spectrogram (for more details read section: 3.1), thanks to `scipy` library (also installed in a layer of this second Lambda) and then sent back the result to the main one. Having the spectrogram, the main Lambda invokes the SageMaker endpoint and sends it, receiving as response an array containing the probabilities that the audio represents the sound of each bird. The main Lambda has the task of selecting the name of the bird with the highest probability and sending as response to the website the name and the probability of the bird in a list.

To optimize the response time, without increasing the costs, we use the tool aws-Lambda-power-tuning, which invokes the Lambda function multiple times with different power configuration levels (from 128MB to 10GB values) and analyzes all the execution logs and suggests the best power configuration to obtain the best performance without increasing the costs. The optimization is due to the fact that we pay the Lambda for the time that it is executed and for the computing power, hence the increase of the computing power may cause a reduction in the execution time. The optimum is reached when the increase of the cost for more power is balanced by a reduction of the cost of the execution time. If the user wants a faster response it is also possible to spend more and get it, but before doing that, it should check whether an increase in the computing power would reduce the response time, in fact it is not always the case, as we can appreciate for our functions).

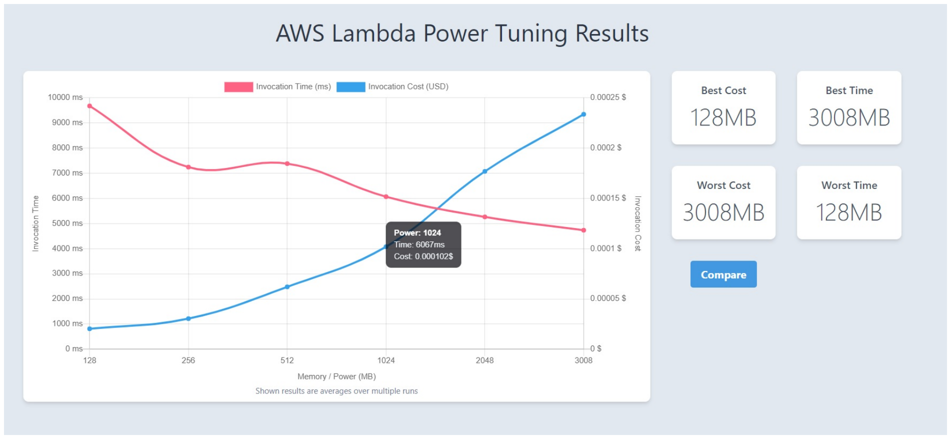

Figure 2: Main Lambda Optimization

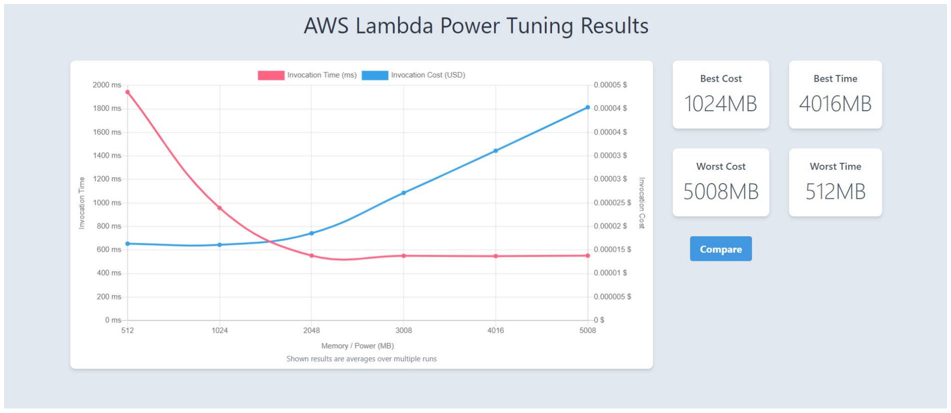

Figure 3: Transform Sound Optimization Lambda

In our case, the optimization doesn't satisfies us in terms of response time, so we have decided to spend a little bit more to reduce the response time between 5000ms and 2700ms (depending on the traffic). For what concerns the first Lambda (Figure 2), the optimization problem gives us an enough good result: 256 Mb of memory, in this situation to have a better time we have to spend too much (the cost increases exponentially). Instead, for the second Lambda (Figure 3), spending a little bit more it is possible to get the best performance available in terms of response time, so we set the memory to 2048 Mb (which is the point in which an additional increase of the power doesn't affect the speed of the execution).

An additional step, to optimize the Lambda function, mainly when there is an important amount of traffic, is to configure provisioned concurrency for our Lambdas. It prevents the latency due to the cold start. However, this reduction implies an extra cost. At the moment we do not expect such a big amount of traffic and we think that a response time between 5000ms and 2000ms is a good one given the constraint of the cost.

2.3 API Getway

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your back-end services (AWS Documentation). We use API Gateway to create a RESTful API that enable the communication with our Web-Application. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls.

In our Web-App we only need a POST method, which allows to transfer into the body of the request the data sent from the resource. When the user sends the request, containing the bird sound, the API Getway triggers the Lambda function that process the JSON file of the request. Before deploying the API, we have to activate the Cross Origin Resource Sharing (CORS), a mechanism that allows to resources (the Web-App and the API) located on different domains to communicate each other. The CORS adds a special header to the packet ('Access-Control-Allow-Origin': '*'), which let the browser to allow the incoming JSON (from the API). Sometimes this is not sufficient, therefore we add this special header to the response directly in the Lambda too.

2.4 Front-end (the Web-App)

Our Web-App is written in Java Script and it is made of one main page(index.html). The page contains a button to upload the audio and a button to send the POST request to the API. Moreover, there is a player that allows the user to listen to what he/she has loaded. The response is printed on the screen, on a line there is the name of the bird and on a second one there is the probability that the sound belongs to that specific bird.

For background a landscape picture is used with flying birds animation. Birds animation is provided by containers (div) and key frames. Containers are used to set animation duration and delay for each move and key frames are used to set bird's position and their moves transforms.

3 Theoretical Overview

The core computation of our Web-App is in the Neural Networks model. In this section a theoretical implementation of the model will be given.
We trained a Neural Networks model to perform a classification. Given the bird sound the objective is to classify the bird's specie. The data in input is an audio of the bird sound and the output is the bird's specie name. Before feeding it into the Neural Networks the audio was processed.

3.1 Audio Features

The first step consisted in the conversion of the audio into a vector representation. An example of an audio waveform is in figure 4.

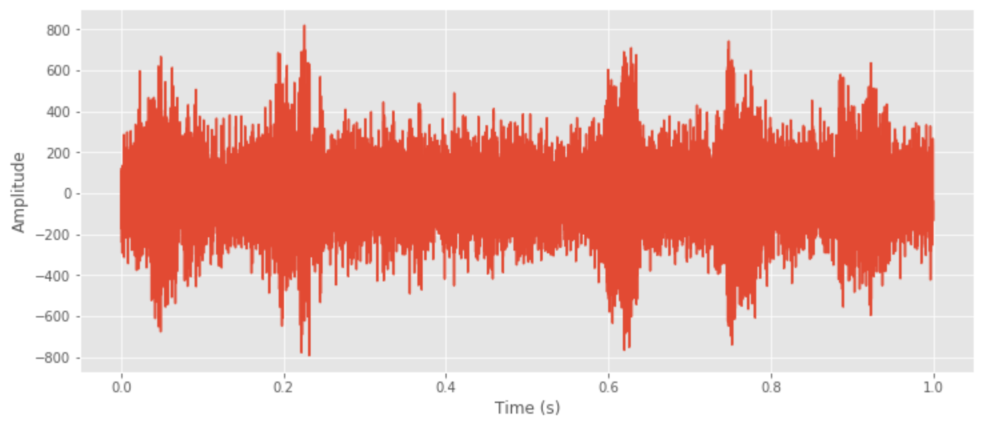

Figure 4: Audio waveform of a sample.

There are several features that one can extract from an audio, but mainly they are split in time features and frequency features. From the first category are available all the features which are related to the time domain of the signal, for example the Root Mean Squared (RMS) that is an averaged and smooth version of the waveform. Meanwhile from the second category there are features that depend on the frequency representation of the signal, such all those that depend on the Fourier Transform. For our project we considered as the main feature the Short Time Fourier Transform (STFT). To better understand the meaning of the STFT it is necessary to introduce the Fourier Transform first.
Given a signal, like an audio waveform, the Fourier Transform is a vector that represents the frequencies that are present in it. From a theoretical point of view a signal with certain characteristics can be expressed as a sum of fundamental functions, each with a different and unique frequency. Therefore The Fourier Transform indicates which of these fundamental functions are present in the signal and how important they are. The STFT instead is a series of Fourier Transforms evaluated on different time intervals of the signal. It is usually represented as a two dimensional plot, where the time is represented on the x-axis and the frequency on the y-axis. An example of STFT is shown in figure 5.

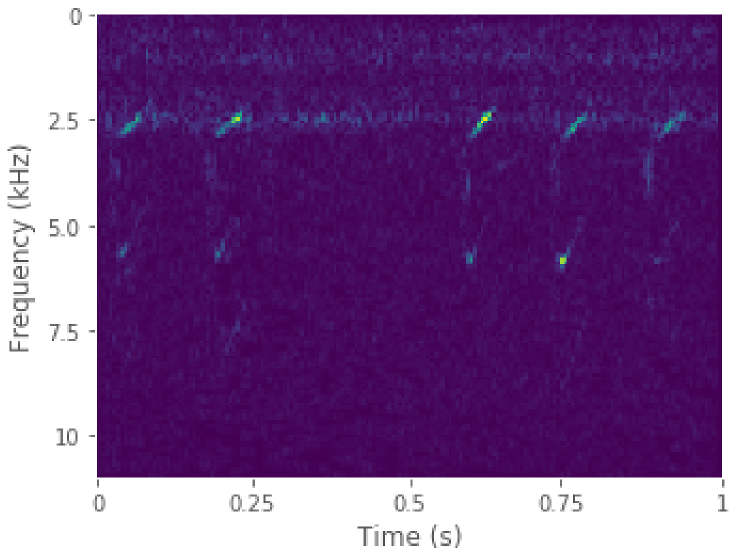

Figure 5: Spectrogram

3.2 Neural Network's Model

The STFT, in practice, is an image with one channel, i.e. a black and white image. Therefore classifying audio with this data is the same as classifying images. The state of the art models for such task are based on Neural Networks.
In our case, we used a simple architecture made of several Convolutional Neural Networks (CNN) blocks. A basic block is made of:

- CNN layer with k channels and (3,3) kernel's dimension.
- Batch Normalization layer.
- ReLU activation function.
- Max-Pooling with a (2,2) window.
- Dropout layer with a parameter of 0.1.

The architecture is composed by 3 of these blocks, with $k = \{64, 128, 256\}$.
Then another block composed by:

- Global Average Pooling layer.

- Dense layer with 256 neurons and ReLU activation function.
- Dropout layer with a parameter of 0.3.
- Dense layer with 128 neurons and ReLU activation function.
- Dropout layer with a parameter of 0.3.

After the last block the model ends with a Dense layer with a softmax activation function.

4 Technical Implementation

The dataset used in this project is available on Kaggle. It is composed by roughly 40 GB of data. The data was loaded into Kaggle by scraping an external website, Xeno-Canto. This website contains audio of bird sounds recorded from amateur users.

4.1 Data Processing

The audio, after the conversion from the `.ogg` format was re-sampled at 22.5 kHz and converted into mono. The model used is a Neural Network that expects in input data with always the same dimensions, for this reason we decided to consider audio of exactly 5 seconds long. If an audio is longer than 5 seconds it is cut into chunks of 5 seconds. If instead one audio is less than 5 seconds the audio is padded by repeating the sound until 5 seconds are reached.
The STFT has multiple parameters such as the number of frequencies to consider, the size of the time intervals or the overlap between the intervals. We have chosen these parameters to achieve a size of the STFT of (216, 514). This means that the length of each interval is 430 samples with an overlap of 50% between the intervals. Furthermore other processing steps were considered for the STFT:

- Take the norm of each element. This is needed to pass from the complex numbers to the real ones.
- Take the logarithm. Needed to give more importance to values at lower amplitudes.
- Rescale all values in the range [0,1]. For the stability of the algorithm and also for the Neural Network's design.

The STFT was computed using the standard Python library `scipy`.
Furthermore, before downloading the audio from Kaggle, the data was filtered by using the provided audio's metadata. We considered only those audio with a length less than 15 seconds, birds with latitute and longitued of the Europe's continent and those bird species that have a total number of audio of at least 100.

4.2 SageMaker

The overall download and processing of the data was executed on a SageMaker Notebook instance.
SageMaker is an AWS service that allows to train and deploy machine learning models. The instance used is a `ml.t2.medium`. On this type of instance is possible to run Python Jupyter notebooks. We used two main notebooks on the same instance, one for the download and processing of the data and one to run the train and deployment of the Neural Networks model.
AWS provides several Python Software Developer Kits (SDKs) that allow to work with many of their services by executing Python code. We used mainly the `boto3` and `sagemaker` SDKs.
The data is download locally to the notebook instance, then it is processed as described in the previous section and then transferred to an S3 bucket through one of the available functions of the `boto3` library.

4.3 Model's training

SageMaker allows to train the Neural Networks models in many ways. The one that we decided to use is by creating a training job. From a practical point of view the training job can be initialized through the `sagemaker` SDK by providing a training script. The training script contains all the code needed to train the model. The SDK allows to train Neural Networks models by using the main Python libraries such as TensorFlow (used by us), PyTorch and MXNet. The script accesses the data provided from an S3 bucket and in output stores the model's artifacts into another S3 bucket. When starting the training job a number k of EC2 instances can be used. In our case we used only one instance of the type `ml.m4.xlarge`. SageMaker automatically starts a Docker container on the EC2 instance for the training of the model. SageMaker fully manages this step, therefore it takes care of creating a container where the training script can run.

4.4 Model's Deployment

After the training, SageMaker saves the model's artifacts into an S3 bucket. It is possible to deploy a Neural Networks model from its artifacts using again the `sagemaker` SDK. When executed the library's function an HTTPs Endpoint is created. The Endpoint is composed by EC2 instances, that in our case were selected to be `ml.m4.xlarge`. On each instance runs a Docker container optimized for TensorFlow Serving. SageMaker creates a RESTful API that accepts JSON requests and returns JSON responses that are needed to communicate with the WEB Server. All instances are able to give responses, and they are all managed by a load balancer.
The user interacts with the Web-App through this API. The request arrives first at the Lambda and from there another request is sent to the endpoint.

4.5 Auto-Scaling

We set an autoscaling policy based on the CPU utilization of the SageMaker's instances. The policy chosen is of the type Target Threshold, i.e., whenever a target value of the CPU utilization is reached, a new instance is added. In our experiment we used mainly two thresholds that will be described in the next section. Moreover we set the base value of the instance count to 1 and the max value to 4. The statistic of the CPU utilization is sampled every t seconds, and the scaling takes place after 3 data points are measured above the threshold. The policy was configured from the SageMaker's SDK and also by editing the Alarm in AWS Cloudwatch. The Scale In and Scale Out cooldowns were set to 300 seconds.

5 Test

The tests were all run on Cloud9 environment using the Artillery.io software, which let us run tests of various length and also modulate the amount of virtual users and requests per second. Artillery offers a wide selection of options to accommodate testing necessities and operates egregiously with AWS.

5.1 Test 1

We decided to test the autoscaling policy of our infrastructure. We set as CPU utilization threshold 70%, with the average statistic sampled every 10 seconds. At the beginning a light warm up phase with 1 request/second was sent to the endpoint for 2 minutes, this phase is seen in figure 6 around the time 11.20, where the CPU utilization is around 20 % and there is only one instance active. From approximately 11.25 until 11.35 the number of requests sent is 5/s ramping up the load. With this amount of requests the CPU load exceeds the threshold, therefore one instance is added to the endpoint. The measured time to scale is approximately 3'24". From the figure at 11.28 the new instance starts working, therefore reducing the average CPU load from 100 % to 50 %. At 11.35 the number of packets sent is increased to 10/s. Again the threshold is exceeded and a new instance is put in, with a waiting time of 3'24". At 11.38 the new instance is operative and the average CPU load decreases again. At 11.45 the number of packets sent is increased to 15/s, therefore again a new instance is put in, with a waiting time of 2'48". After 15 minutes that the working load is below 60 % the instances starts to scale down, with a cooldown of 5 minutes.
The scaling policy acted as expected and in general our infrastructure worked well in this test. The overall test lasted about 30 minutes and a total of 18120 requests were sent. All the requests sent had a response with a 200 code. The latency was approximately constant for all the duration, with a

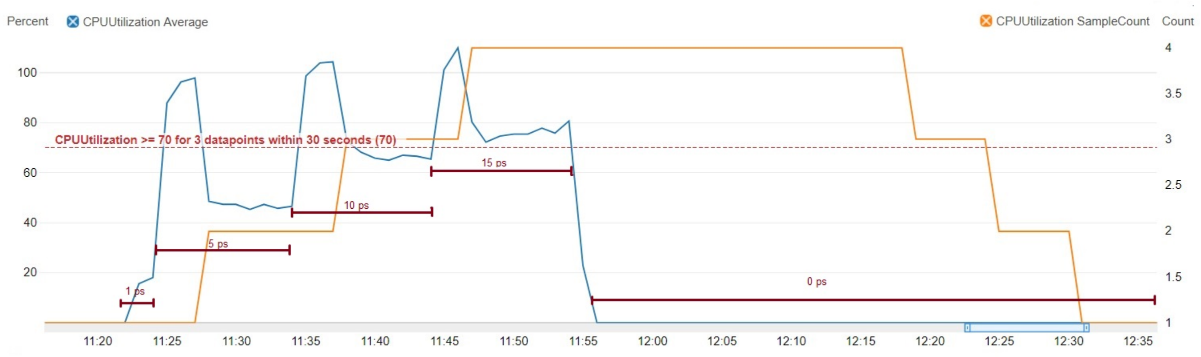

Figure 6: Test of the autoscaling policy. In blue the average CPU utilization across all the instances and in orange the number of active instances. In bordeaux, for each time instance, the number of requests per second sent to the endpoint. Data provided from AWS Cloudwatch.

median value of 1838 ms, a minimum of 1810 ms and a maximum of 1908 ms.

5.2 Test 2

We performed another test sending a constant number of 20 requests/s. This time we set as threshold for the autoscaling 300 % with a statistic sampled every minute. The result is shown in figure 7.

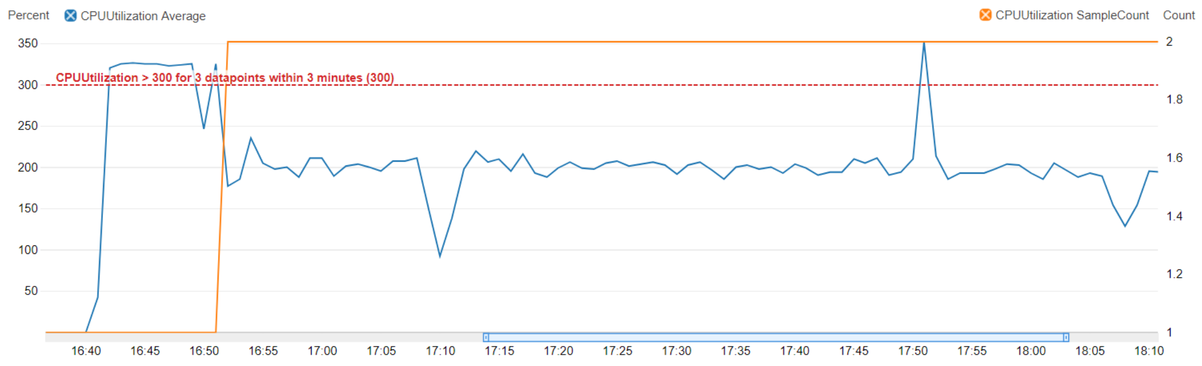

Figure 7: Second test. The number of requests is constans at 20/s. In blue the average CPU utilization and in orange the number of active instances.

The reaction of the infrastructure to this type of load is acceptable. Approximately 100000 requests were sent, all responses had code 200. The median latency was 1837 ms, with a minimum value of 1823 ms and a maximum value of 1851 ms.