Cloud Computing Project Report - XRay Xaminer™

Wednesday, May 12th, 2021

Group Members: Catherine FitzGibbons, Bridget Hart, and Dominic Marques

Overview of goal or problem

The purpose of this project is to scale up machine learning and be able to deploy the ML model at scale in a distributed environment for use by the general public. Our project takes an existing machine learning task, predicting medical conditions based on chest X-rays, and accelerates it on a distributed system. The dataset to train our models consists of 112,120 768x768 chest X-rays images and was found on Kaggle, as was our preexisting model. Each image is classified as having 0, 1, or more of 14 possible medical conditions. As an interface for our project, we created a webpage that allows a user to submit a chest X-ray and have the model run on the image. The diagnosis predictions from the model are then relayed back to the user in a timely manner.

The final goal of this project is to provide a non-official screening service based on the machine learning model, to help provide initial diagnoses to be followed up on with a more experienced physician. A secondary goal is to find the optimal amount of training data to use in order to speed up the training process of the model, while still allowing for a reasonable amount of accuracy.

Detailed description of the structure of your system and the application

The nature of this project meant that there were two separate systems which, while they did share information, did not interact directly with each other. The first of these systems is the model training architecture, which is based on HTCondor.

Model

Because this project was a cloud computing project and not a machine learning one, an existing CNN and model code was used instead of creating a new one. The model took files in the TFRecord format, split them into training, testing, and validation sets, ran five epochs of training, and then evaluated for accuracy. The model code used the Tensorflow framework with the Keras API to train, validate, and evaluate the resulting model. Upon running the code, the user received an output of the model, the time to train and evaluate, and the AUC measure. The AUC measure is a metric for machine learning models that represents the area under the ROC curve which correlates to how accurate the model is with new images.

Model Training

______Initially, the model was run locally to yield a baseline execution time and a version to interface with the front end application. Once the model was functional from a local machine, speeding up and making the training more robust was the next task. Initially, distributing the training of one model was considered. However, the Keras API did not interface well with the HTCondor system. There was the functionality to distribute the training, but the program had to know the addresses of the machines it would be running on rather than picking machines out of a cluster. Beyond the difficulties of interfacing Keras and HTCondor, the training itself did not take so long that distributed training was necessary in order to complete the computation. The model is also dependent on quite a few python packages that take time to install. Installing all the dependencies before running took time and might make distributing the training less efficient.

Instead of distributing one model's training, the training of multiple models was distributed. The amount of data used for training was varied to see how the accuracy of the model and time to train interacted with each other. Multiple models were trained at each percentage of training data so that the average accuracy could be observed. To implement this with the HTCondor system, both a current version of Python and the necessary packages (such as pandas, sklearn, tensorflow, and keras) were saved as a tarball. These tarballs were included as input files in the condor submission script. The x-ray data files were also included as input files along with the model code. The executable for the submit script was a shell script that first installed python and the related dependencies, then ran the model code. After the job ran to completion, HTCondor returned the trained model, the time to train, and the AUC measure.

The model training architecture can be seen in the diagram below:

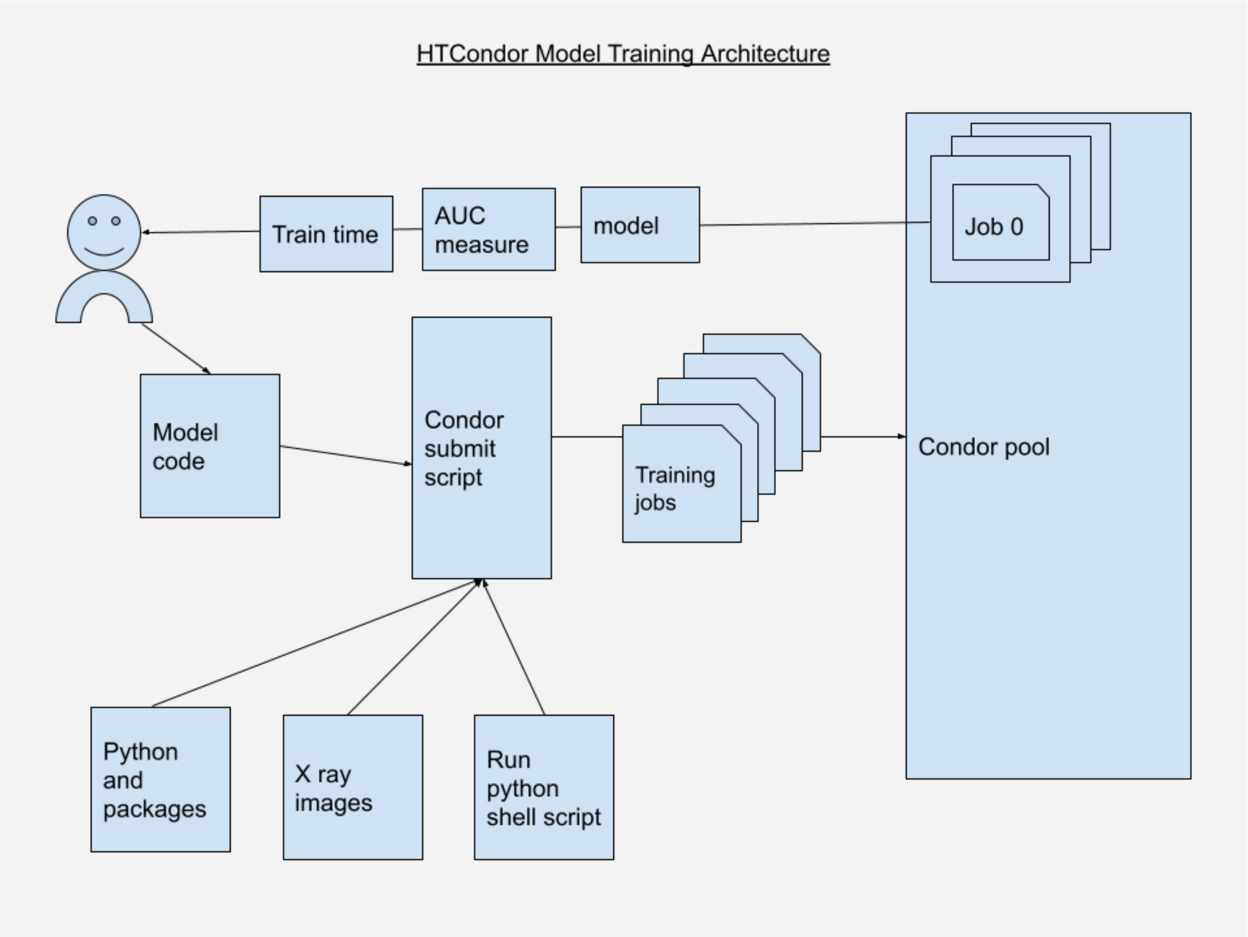
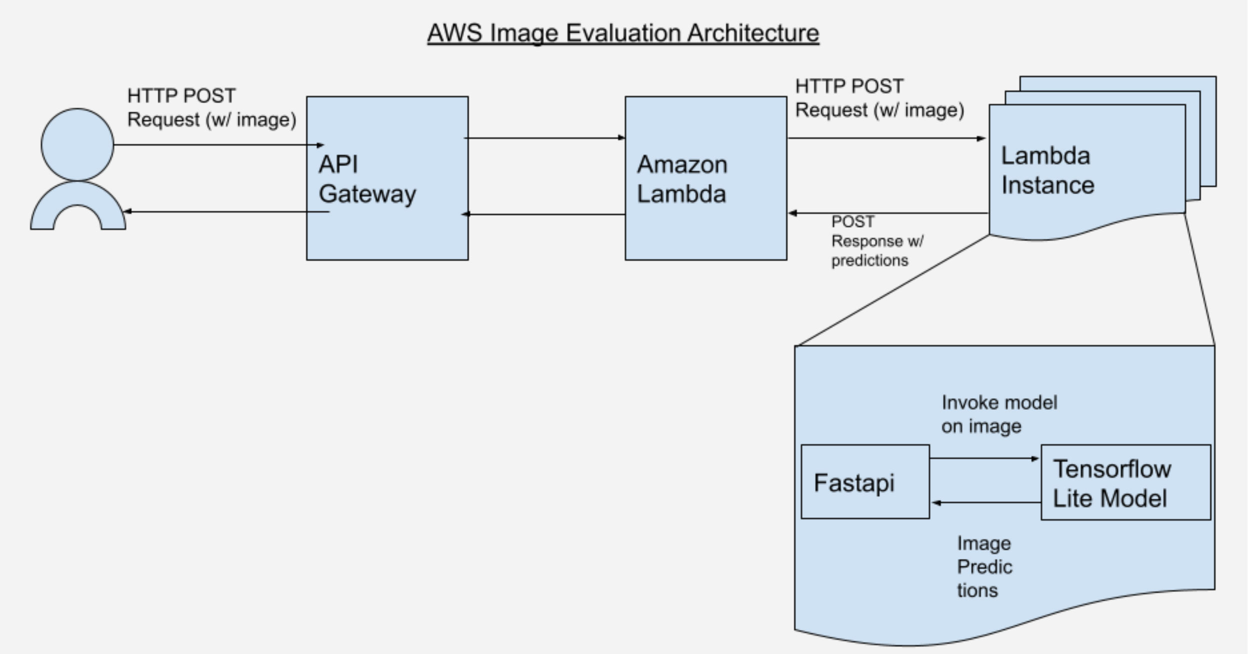


Image Evaluation

The second system is the image evaluation system, which is built on AWS. This system is built on a combination of FastAPI, which is running on Amazon Lambda. The api is exposed to the internet using API Gateway. The API Gateway will route requests coming through the endpoint directly to a Lambda function running FastAPI, which will in turn load the saved model and run the image through for evaluation. The result will then be returned to the user as the

HTML response to the POST request. The architecture can be seen in the image below.


AWS Image Evaluation Architecture
HTTP POST Request (w/ image)
API Gateway
Amazon Lambda
HTTP POST Request (w/ image)
Lambda Instance
POST Response w/ predictions
Invoke model on image
Fastapi
Tensorflow Lite Model
Image Predic tions

This website consists of just one GET and one POST method. The GET method is the initial path, which returns a simple HTML form for submitting the image for evaluation. The POST method takes in the image which would be used for evaluation, runs the evaluation, and then returns the results of the evaluation in HTML form.

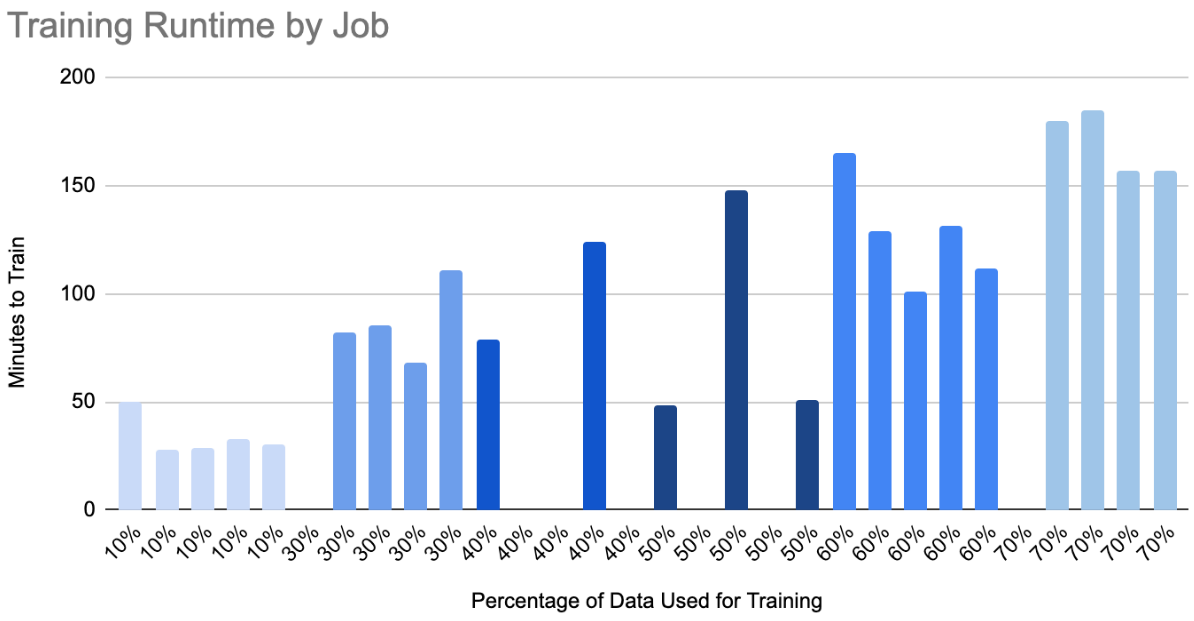
The model used for the image evaluation is not the exact same model that is created above. There is a bit of processing of the large TensorFlow model created by the model training that is required in order to shrink the total package size of the FastAPI deployment package. Amazon Lambda has a maximum unzipped package size of 262 MB which can be run as a single function instance. In order to shrink the deployment package to be small enough for deployment on Lambda, the TensorFlow model had to shrink, as did the number and size of the libraries included for the model to be able to run. In order to do this, the original model was first converted to a TensorFlow Lite model. TensorFlow Lite is a Python framework that is based on TensorFlow, but is optimized for mobile and IoT devices. This allows the sizes of the required packages, as well as the model binary, to be very small. It also comes with the added benefit of utilizing significantly fewer system resources. This then reduces operational costs of each Lambda function call, which is billed on a combination of execution time and memory usage, both of which are reduced.

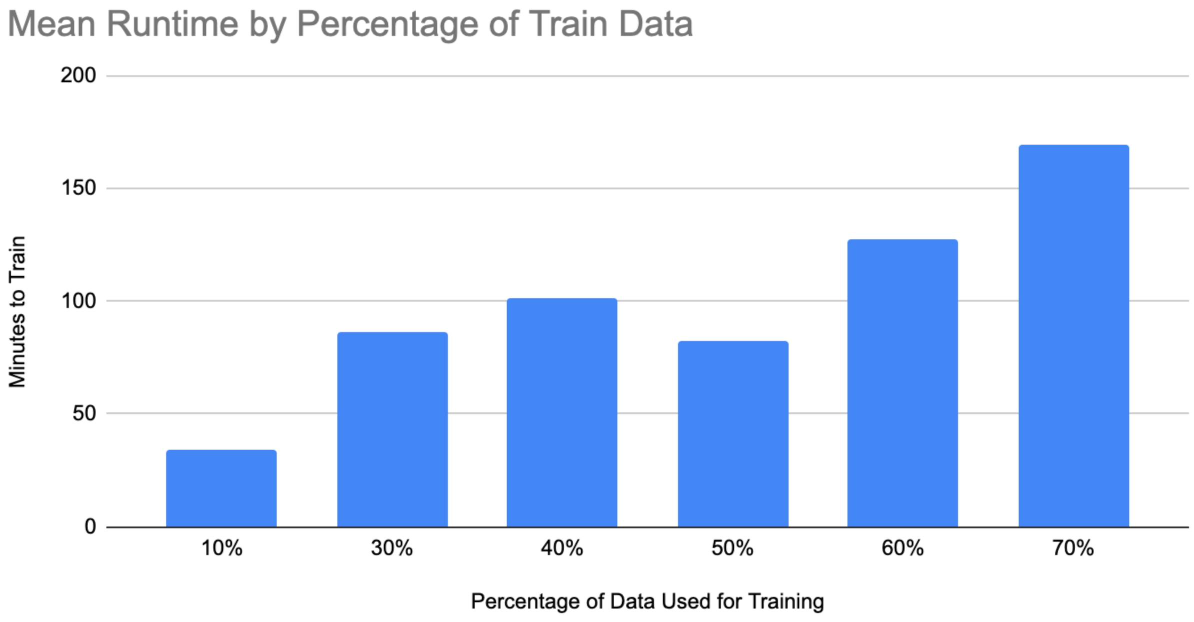
Evaluation of correctness and performance of the system

Model Training Evaluation

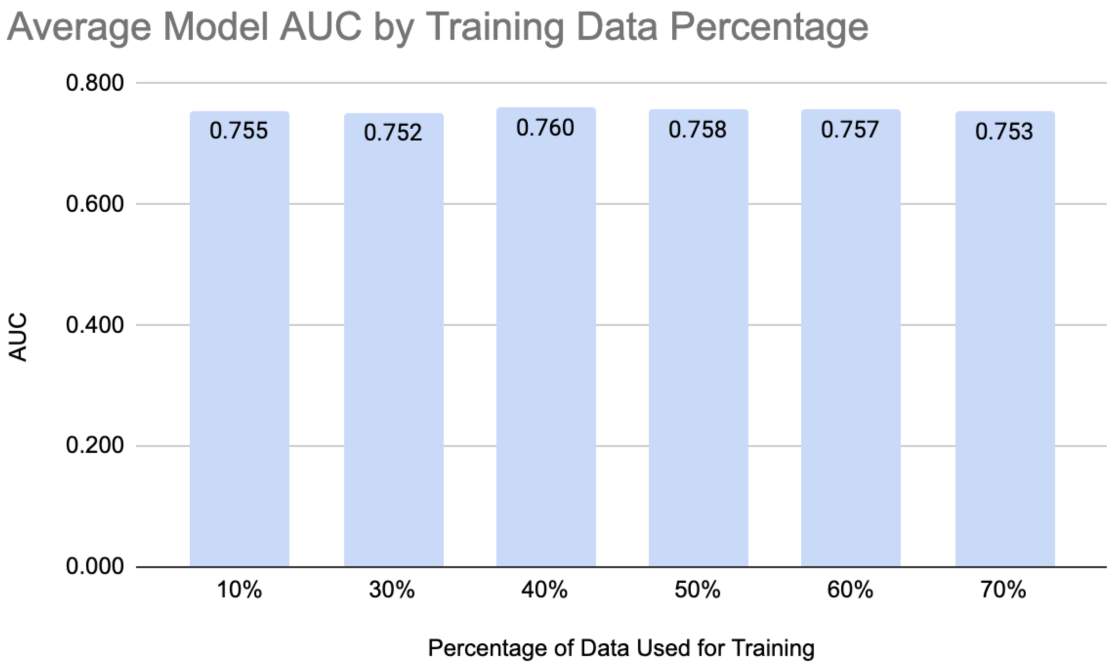
The two graphs below show the raw and average data for the amount of time to run the model training code remotely. The independent variable is the percentage of available data used for training. Both graphs show the trend that as the amount of data used in training increases, so does the execution time for the model. This makes sense because the amount of data used in training changes the steps per epoch, even though the number of epochs remained the same throughout each trial.

As seen in the raw data graph below, there are some training jobs that did not run at all. This was due to machines not having the correct hardware for the Tensorflow framework and returning an “Illegal Instruction” error. This type of error makes sense in a cluster of diverse machines where they are not all the same level of newness or have the same features. It also reinforced running multiple trials at each training level because not all of the jobs were going to be successful. The graph below does not have any data from the 20% split because all five trials at that level returned as errors. In the future, adding more requirements to the HTCondor submit script about what type of machines to match with could be a way to avoid failures due to this error.


Training Runtime by Job


Mean Runtime by Percentage of Train Data

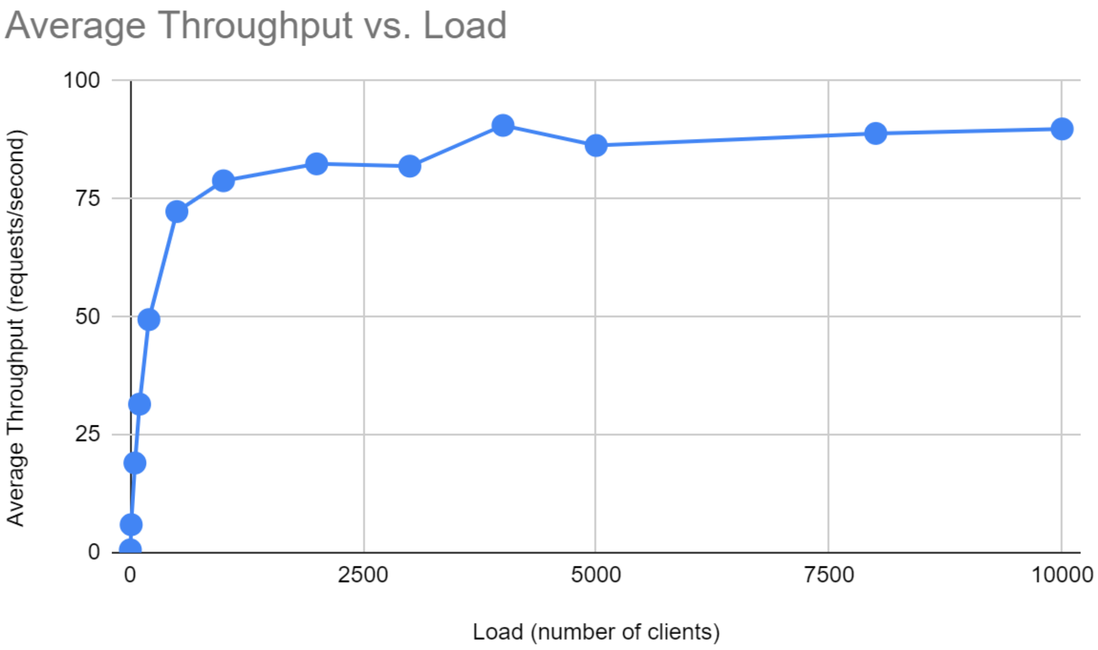
The graph below shows the average AUC measurement for each data split. AUC measures range from 0 to 1, with a higher AUC meaning the model makes better predictions. There is not a noticeable difference between the AUC for the different training splits. This might mean the model itself needs improvements, or that 10% of the data is sufficient for training and the additional images are not necessary. If the latter was the case, a future extension of this project would be to test below 10% and identify what the minimum number of training images is for an AUC of 0.75. However, in the specific case of our model, the consistent AUC measurements result from the highly imbalanced dataset the model is trained on, which, on average, classifies almost 95% of x-ray images as not having a given condition. This imbalance is important to observe because at first glance the dataset appears to be split 54% and 46% for x-ray images having and not having a condition, respectively. However, this does not reflect the entire dataset where the majority of the 54% findings consist of one true out of fourteen, leading to the true imbalance. Due to this, and the high confidence level the model requires to predict an x-ray as having a condition (50%), the model learned to predict false for every condition, no matter the x-ray image.This explains the consistent AUC measurements, as each model predicted roughly the same number of true negatives and false negatives in proportion to the distribution of the dataset.



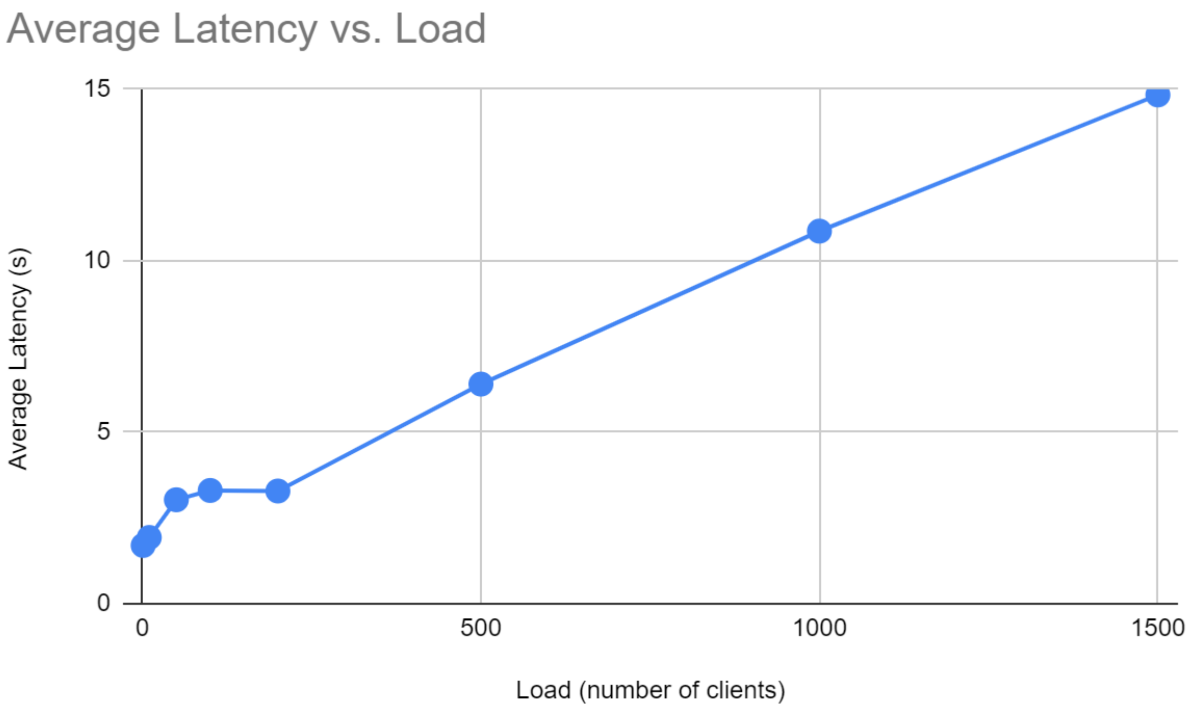
As a model that always predicts ‘no condition’ does not have a practical application, to add functionality, the thresholds for the 14 conditions were lowered so that the model does not always predict no. This results in the model predicting a mix of true positives, true negatives, false positives, and false negatives, and therefore acts more as an initial screening for possible conditions rather than attempting to give an exact diagnosis. This tradeoff for the added functionality, which brings the precision and recall values above zero, is the significant increase in false positives, however this is expected given the goal of our project. The lowered thresholds were calculated by taking a subset of predicted probabilities for each of the 14 conditions given by the model, categorizing each probability as having or not having the condition according to the known diagnosis, and attempting to find a split between the probabilities corresponding to what the model predicted when an x-ray had a condition and when it did not have a condition. In the subset of data an exact split was not possible, so to err on the side of caution, the lowest probability for when an x-ray has a given condition was chosen as the threshold.

Website Evaluation

The evaluation of the website’s scalability focuses on increasing the load, or number of users uploading an image for prediction, on the website. This was tested using async curl requests to our website in a bash script and calculating the throughput as requests/second. The scripts were all run multiple times, as to get the average values. The throughput was measured for 1 client, up until 10,000 clients. In the graph below, the productive limit is seen to be around 1,000 users. A critical value, or the point at which the load becomes too much to handle and throughput rapidly decreases, was not encountered. Given the fast execution time per function call of our website, we believe the critical value is well past 10,000 clients and that our project scales well on Lambda.



The performance of the website can also be evaluated on the time it takes to predict the possible conditions of an x-ray image uploaded to the site, or the latency. Given the throughput calculations, it was anticipated that latency would remain relatively constant before 1,000 users and then increase with additional clients past 1,000. However, our calculations, also measured using asynchronous curl requests in a bash script, did not yield these results. Rather, the latency hovered just around or below 3 seconds for 200 users or under, and then began to steadily increase with the load after. These results could be due to the way latency was tested and more accurate results could be yielded from a load testing service such as Artillery.io. Another possible explanation is that there is a bottleneck with api gateway, which is inhibiting the performance Lambda makes possible.



Overall, the website successfully takes in and predicts the medical conditions of a given x-ray image and scales well to an increase in load. There is room for improvement in its performance, and this would be something to investigate as a future extension of this project.