Tumor Detection – Machine Learning Deployment

(Shourya Narayan, Pranjal Jain , Manthiramoorthy Cheranthian, Shreyak Bhattacharya)

Project Overview

Machine Learning Model deployment is one of the most difficult processes of gaining value from machine learning. It requires coordination between data scientists, IT teams, software developers, and business professionals to ensure the model works reliably in the organization's production environment.

Recent machine learning (ML) methods have shown accurate predictive ability and have been increasingly used in the diagnosis and prognosis of various diseases and health conditions. ML algorithms like Random Forest have been applied to detect key features of patients' conditions and to model disease progression after treatment with complex health information and medical datasets. Meanwhile, studies have shown that ML models can be constructed with sex, age and complete blood cell count data to detect early colorectal cancer. By deploying our ML we will be able to classify cells to whether the tested samples are benign or malignant and will help the patients in early detection of tumor.

Below mentioned figure depicts the workflow of ML Model and its deployment into production.

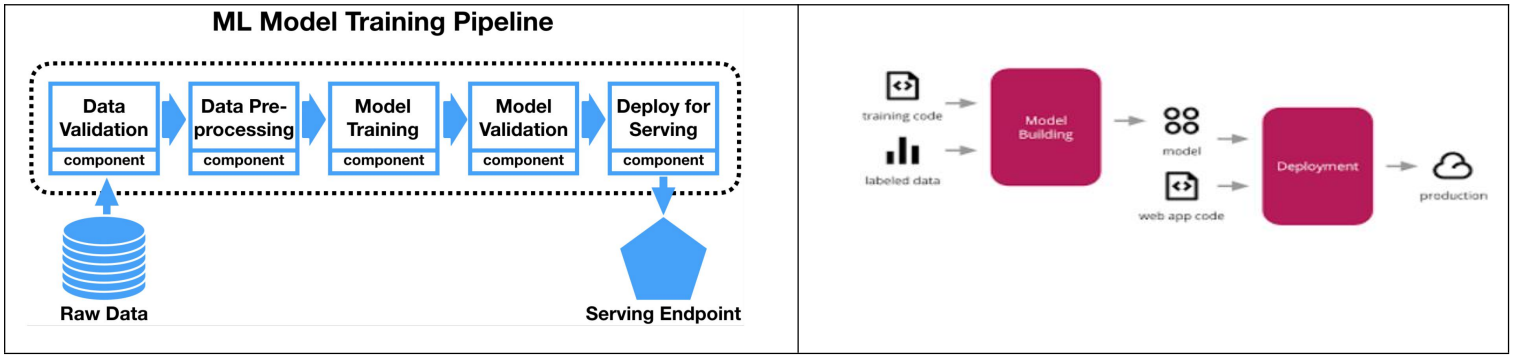


Figure 1

Cost Analysis:

AWS Professional services provide best prices for businesses of all sizes. We can start our application with minimal needs, can easily scale our application as our business grows up.

AWS EC2 usage are billed on one second increments, with a minimum of 60 seconds. Similarly, provisioned storage for EBS volumes will be billed per-second increments, with a 60 second minimum. Per-second billing is available for instances launched in: On-Demand, Reserved and Spot forms

Certain Amazon SageMaker resources (such as processing, training, tuning, and batch transform instances) are ephemeral, and Amazon SageMaker automatically launches the instance and terminates them when the job is done.Therefore, knowing how to identify idle resources and stopping them can lead to better cost-optimization.

The Total Cost of Ownership (TCO) is often the financial metric that you use to estimate and compare ML costs. This post presents a TCO analysis for Amazon SageMaker, a fully managed service to build, train, and deploy ML models. The findings show that the TCO over a three-year horizon is 54% lower compared to other cloud-based ML options such as self-managed Amazon EC2 and AWS managed Amazon EKS.

Steps implemented-

- Developed a model API using Flask/ Django, a web microframework for Python.
- Deployed the model API to the web using AWS EC2.

A web developer has option to choose from a wide range of web frameworks while using Python as server-side programming languages. He can take advantage of the full-stack Python web frameworks to accelerate development of large and complex web applications by availing a number of robust features and tools. Likewise, he can also opt for micro and lightweight Python web frameworks to build simple web applications without putting extra time and effort. Both Django and Flask are hugely popular among Python programmers. Django is a full-stack web framework for Python, whereas Flask is a lightweight and extensible Python web framework. Django is developed based on batteries-included approach. It enables programmers to accomplish common web development tasks without using third-party tools and libraries. Django lacks some of the robust features provided by Python. We have used both frameworks (Djano & Flask) in our project and then deployed the ML model using AWS EC2.

Django-

Pickling the Model:

We use joblib library to dump and pickle the built model in .sav format such that it could be transferred into our Django project and loaded there for usage by our python functions.

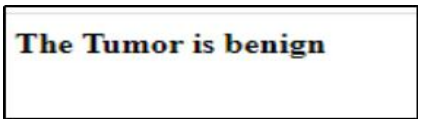
Building the Templates:

Index.html

Building a basic HTML page with number of text fields based on the number of input fields required for the model to predict. We use 9 cell characteristics as inputs to be received from the user. We use a “Predict” button which calls the result function in the views.py. We also use CSS and some JavaScript to make the page responsive.

Result.html

This is a basic HTML page just to show the result of the tumor. We use jinja tags to use if else within HTML tags. The value of 'ans' is carried along with the request from the home page.



Django ML Application:

Step 1:

Create a Django Application named tumorclass using the django start project command

'Django-admin startproject tumorclass'

Step 2:

Creating a folder named Templates and storing the two HTML files inside the Templates folder which could be easily accessed by Django.

Step 3:

Saving the extracted model file inside the project folder in the root directory itself to be accessed within the Django by the functions in the views.py file.

Defining the functions to load in views.py

Function 1: home

This function returns the index.html page from templates whenever the IP address is being called.

Function 2: result

We unpickle the model using joblib.load command and load it inside a storing variable for the model. We also create an empty list named 'lis' and append the input values received from the home page to the list. Now we create a new variable which stores the predicted value '2' or '4' using the model. Finally we render the result.html which along with passing the predicted variable to the page where we could process it using jinja code.

Step 4: URL Mappings

Mapping the path and URLs to the functions which we created in the views.py file. Whenever user calls just the IP address or the localhost in his system the homepage will be returned whereas localhost/result will call the result page along with the input variables.

Since we created a distinct app under the project, we are including the URLs paths of the app under the main project.

Step 5:

Mapping the templates to the Base Directory, For Django to find the HTML template to render we are mapping the Templates path in the settings.py file under Directories

Step 6:

Starting the Django Server:

Now finally once the app is built and the configurations are set, we start the Django server such that the application starts running on the local host IP address. We use the following command to run the application 'python manage.py runserver'.

```
C:\Users\manth\ids594\tumorclass>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations
ons.
Run 'python manage.py migrate' to apply them.
October 13, 2020 - 12:44:48
Django version 3.1.1, using settings 'tumorclass.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Deployment Strategies:

Now the application is built and running we implement following options to deploy the model into the cloud.

1.) Deploying the application into the AWS EC2 Linux server
2.) Deploying the application into AWS ECS using Docker.

Deployment Strategy One:

Deploying the application into the AWS EC2 Linux server

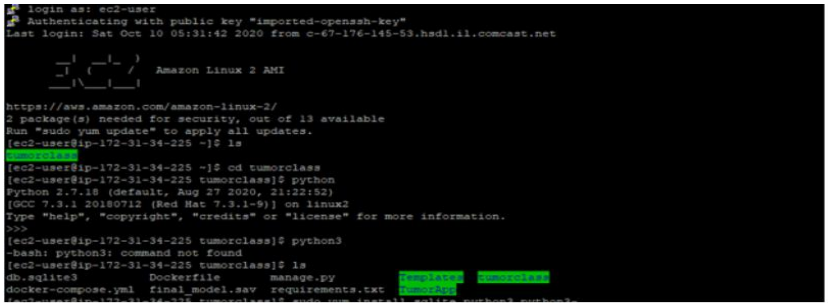
Create an AWS Linux EC2 instance

Deploying the application into the AWS EC2 server:

Step 1: Pushing the entire Django Application into a Linux EC2 server using CMD prompt in our local machine

```
C:\Users\manth\ids594>scp -r -i  ~/Downloads/ecs594.pem tumorclass ec2-user@35.154.149.213:
db.sqlite3                                    100%    0     0.0KB/s   00:00
docker-compose.yml                            100%  200     0.2KB/s   00:00
Dockerfile                                    100%  501     1.6KB/s   00:00
final_model.sav                               100% 1032KB   1.0MB/s   00:01
manage.py                                     100%  680     3.0KB/s   00:00
requirements.txt                              100%  205     0.9KB/s   00:00
index.html                                    100% 4004    17.3KB/s   00:00
result.html                                   100%  173     0.8KB/s   00:00
admin.py                                      100%   66     0.3KB/s   00:00
apps.py                                       100%   96     0.4KB/s   00:00
__init__.py                                   100%    0     0.0KB/s   00:00
models.py                                     100%   60     0.3KB/s   00:00
```

Now we can see that after push all the files are visible inside the linux EC2

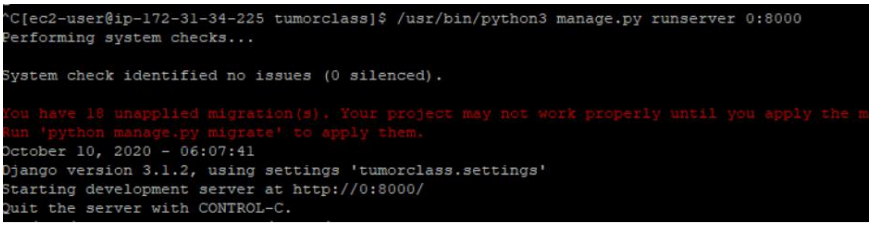


Step 2: Install Django inside Linux EC2

Step 3: Install all the requirements for the app in the EC2 (Scikit, joblib, numpy etc)

We can get the requirements by using pip freeze requirements.txt inside the Django app

Step 4: Use the following command to start and run the Django server to run from the EC2 machine. We also define the port as 8000 where our application will be running. /usr/bin/python3/ manage.py runserver 0:8000



Step 5: Now we can run the IP address of the EC2 in our browser followed by the port

Command: http://35.154.149.213:8000/

Deployment Strategy Two:

Docker and Docker Compose file:

We create a docker image inherited from python 3.7 where we are exposing the port 8000 and creating a directory named /app where we are adding all the Django files which we created. We are installing all the required libraries and dependencies inside the container such that the container environment becomes able to run our Django application. In the final command line, we are providing the run server command in the docker file which initiates the Django server in the container once the container is started. In the Django compose file we map the port 8000 of the local machine to the world. We also configure the security groups in AWS Linux in such a way that it allows HTTP, HTTPS, and Custom TCP connections from the user.

Building the Docker Image:

In this step we are calling the Docker-compose .yml file to build the docker container which downloads and installs all the libraries specified in the requirements.txt file and builds the docker image.

```
Step 10/10 : CMD [ "python", "tumorclass_web/
 ---> Running in ec2c18f6a34e
Removing intermediate container ec2c18f6a34e
 ---> f0c03f8cad7f

Successfully built f0c03f8cad7f
Successfully tagged tumorclass_web:latest
```

Tagging the Docker Image and Publishing it to Docker Hub:

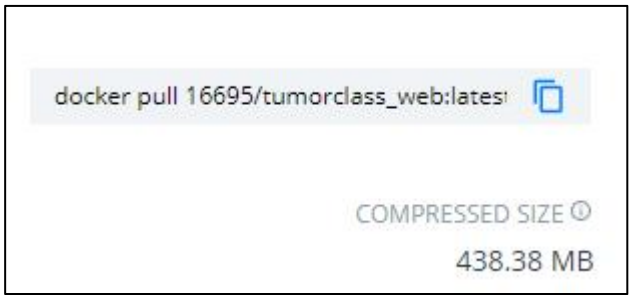
In this step we tag the built docker container to an image name named tumorclass_web, Once the image is tagged to a name, we are publishing the image to the docker hub. We create an account in the docker hub and publish the image to docker hub.

```
C:\Users\manth\ids594\tumorclass>docker tag tumorclass_web 16695/tumorclass_web

C:\Users\manth\ids594\tumorclass>docker push 16695/tumorclass_web
The push refers to repository [docker.io/16695/tumorclass_web]
```

Pulling the docker image to the ECS optimized EC2 Linux machine.

In this step we create a ECS optimized EC2 Linux machine and configure the security groups to allow requests to port 8000 of the Linux Machine through the EC2 IP address. Once the ECS machine is started and running we pull the docker image from the docker hub and start the container to run in the ECS machine.



FLASK-

Creating a virtual machine-

- The virtual machine has been created using AWS EC2 instance with ubuntu 20.0 deployed on it.
- The ubuntu 20.0 and the virtual machine contains relevant ports for two-way communication that is inbound and outbound
  - These ports help the machine to communicate with the external web eco system using protocols such as HTTP, HTTPS, TCP, SSH.
- This machine can be accessed from the local windows environment using SSH protocol.
  - The tools used to communicate using SSH are putty (to access terminal) and filezila SFTP client(to transfer file using SFTP)
- The virtual machine has been equipped with the required dependencies for the flask app to run without any anomalies.

Storing the trained model -

- The model has been trained on the colab ipython environment provided by google and then stored in a flat file using joblib api.
    - This trained model is downloaded in the local environment from colab and further uploaded to the virtual machine created previously.

Creating a flask app-

- A flask app has been created to utilize the trained model via an appealing user interface.
- This flask app contains the main program in the root directory and corresponding HTML and CSS files in there respective reference location for the user interface.
- Working of the app:
    - This app can be used from the web to determine the type of cancer by giving relevant inputs in the user interface.
- This app runs as a background process in the ubuntu virtual machine using nohup.

Machine Learning Deployment with Amazon SageMaker

With Sagemaker, Deploying an ML model is comfortably an easy process.
We can traditionally re-engineer a model before you integrate it with your application from notebook instances and deploy it.

Deploying a model using SageMaker hosting services is a three-step process:

1. **Create a model in SageMaker** —By creating a model, we tell SageMaker where it can find the model components. This includes the S3 path where the model artifacts are stored and the Docker registry path for the image that contains the inference code.

2. **Create an endpoint configuration for an HTTPS endpoint—** We specify the name of one or more models in production variants and the ML compute instances that we want SageMaker to launch to host each production variant.When hosting models in production, we can configure the endpoint to elastically scale the deployed ML compute instances. For each production variant, you specify the number of ML compute instances that you want to deploy. When you specify two or more instances, SageMaker launches them in multiple Availability Zones. This ensures continuous availability. SageMaker manages deploying the instances.
3. **Create an HTTPS endpoint**—Provide the endpoint configuration to SageMaker. The service launches the ML compute instances and deploys the model or models as specified in the configuration. To get inferences from the model, client applications send requests to the SageMaker Runtime HTTPS endpoint.

For further elaboration we have also added another document having all the stepwise - screenshots of the ML model deployment via AWS EC2.

RESULTS:

Developed tumor detection model hosted and available in the specified IP address of the EC2 Linux machine.

Simple and light web application in cloud with no database helps in high responsiveness.

Django Webserver is used since this is a simple application whereas we could also use Gunicorn server for handling production level requests.

Docker helps us in containerization of the application and publish the docker image in the docker hub repository. Any user could pull the docker image and make the application running instantly.

AWS Sage maker reduces the cost of running an application in the cloud significantly and with efficient performance.