Deploying a recommender system on the cloud

Emilio Macias · Follow

9 min read · Jul 1, 2021

16

Photo by Felix Mooneeram on Unsplash

Abstract

This project covers the end-to-end process of training and deploying a recommender system on AWS, following a typical use case such as predicting movie ratings from users. Although a basic introduction to recommender systems is given, the mathematical reasoning is out of the scope of this article, focusing primarily on the practical side of model building and service deployment.

Background

Gone are the days when we would visit a video store to get some movies to watch. Users now have access to a great deal of movies with the click of a button. However there is something that they lack more and more everyday, and that's spare time. Most users are not willing to spend an hour searching for the ideal movie to watch at a certain time. Personal recommendations disappeared when the video stores closed.

Thanks to software and statistics, this problem can be solved with a so-called *recommender system* which is nothing but a machine learning system that can predict users' interests and ratings they would give to an item with the aim of helping the users discover new products they might enjoy. Basically the more data we have collected about an user (*movie ratings*) the better we know their preferences and so the more accurate the *recommendations* will be.

Photo by Sean Benesh on Unsplash

Recommender systems

Coming back to the age of movie stores, the store assistant would come to you and ask you for your preferences, maybe they'd ask you for your favourite movies and recommend similar ones that you might not have watched. This is what we would call *content-based* recommendations since they are only based on the similarity of the items, and that can be easily solved with recommender systems.

Another way of recommending movies is called *collaborative filtering* by which items are recommended based on user profiles. Imagine the scenario where a store assistant would personally know everyone that visits their store, including their interests. When customer X arrives at the store, the assistant knows customer X has similar preferences to customer Y and so he/she will recommend movies that customer Y enjoyed and haven't been watched by customer X. As you can imagine, this approach was not feasible from a human perspective, however it can be solved with linear algebra and matrices by collecting rating and watching information in order to find similarities between users.

One type of collaborative filtering systems are model-based algorithms which make use of machine learning methods to find these user similarities. For this project we are using an algorithm called Singular Value Decomposition (SVD) which is available in scikit-surprise, a Python library for recommender systems.

If you are interested in the mathematical details of these methods I suggest you reading the Introduction to Recommender System.

ML Pipeline

One of the most common ways to deploy an AI service is through an API that can receive HTTP requests and uses a machine learning model to make predictions (in this case, a rating from a given user for a particular movie). Here we are going to build a ML pipeline using different services from AWS.

Below is the list of AWS services and external tools we'll be using for our pipeline:

- Flask, Pandas and Scikit-Surprise (Python libraries) for coding the ML algorithm.
- Docker for packing our algorithm into a container.
- Amazon S3 for storing the datasets and the trained model.
- Amazon ECR for hosting our custom algorithm in a Docker container
- Amazon SageMaker for training the model and for making predictions
- AWS Lambda function for invoking the SageMaker Endpoint and enriching the response.
- API Gateway for publishing our API service to the Internet.
- Amazon CloudWatch for event logging.
- Postman for sending HTTP requests to our public API.

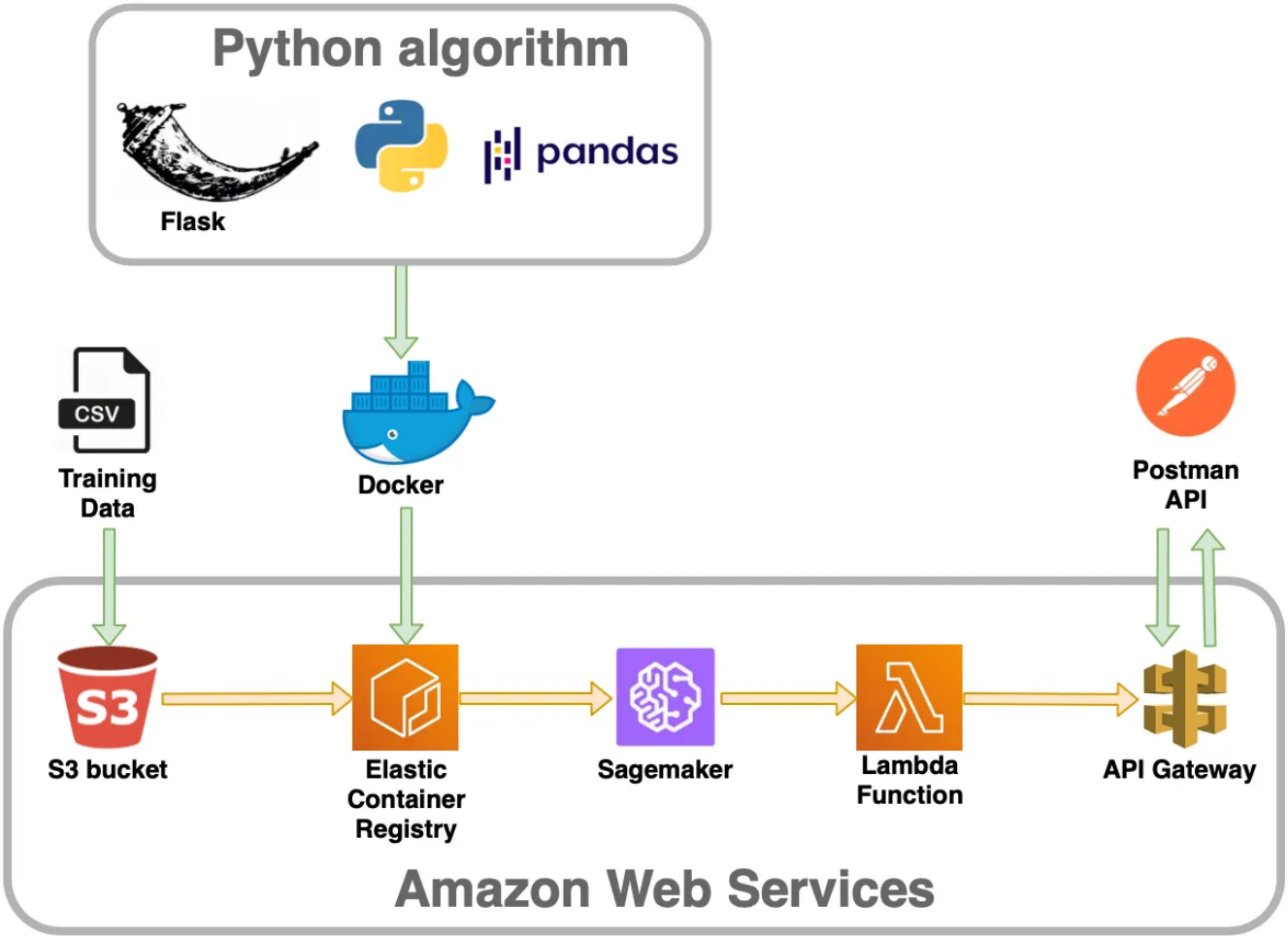


Image by Author

Preparing Data

The first step is to collect the data that we need for our recommender system. In particular, we are interested in movie ratings information that will help our model find similarities between movies and users. One of the largest movie databases is MovieLens, a dataset collected by the GroupLens Research and available here.

AWS provides a service named S3 where we can upload our datasets into object buckets that can be accessed via other AWS services, custom models or even containers.

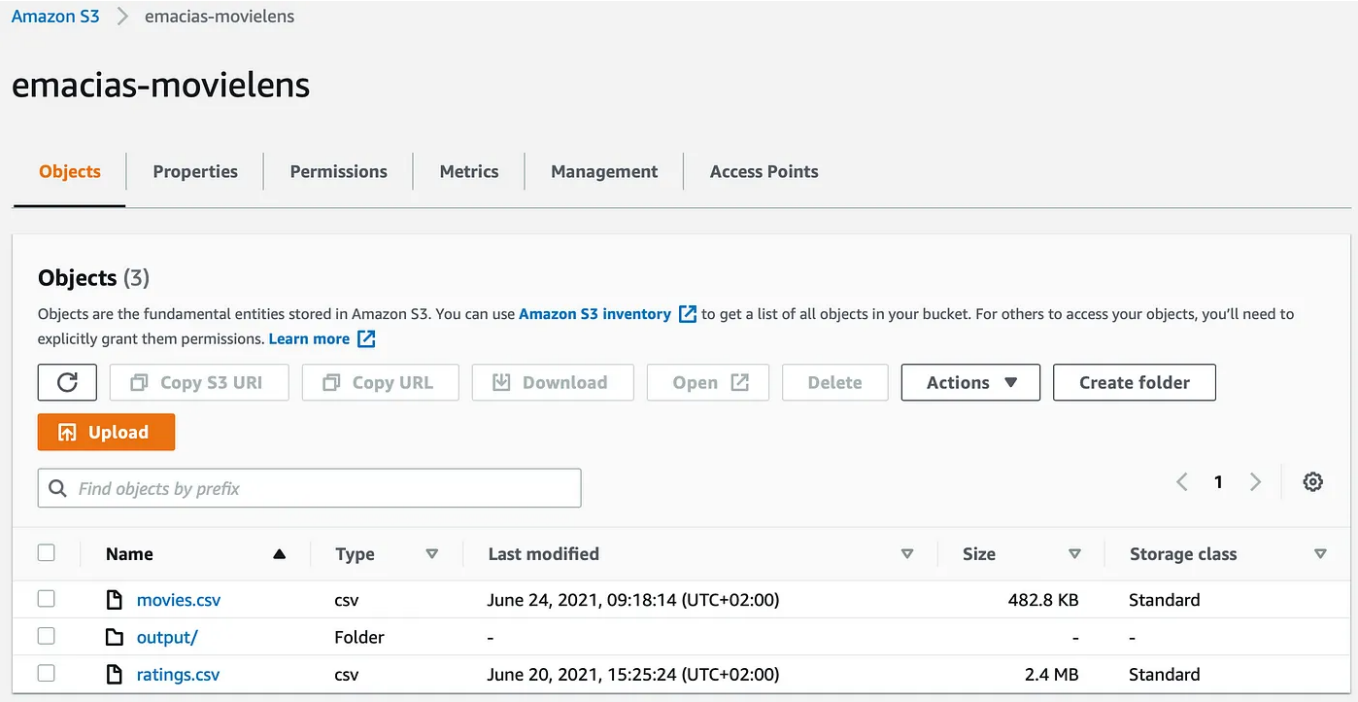


Image by Author

Containerising our algorithm

Even though SageMaker provides a large list of built-in ML algorithms, building your own custom algorithm will provide you better flexibility for particular scenarios.

AWS expects your ML algorithm to have a *train* and a *serve* component, as explained in this post: https://aws.amazon.com/blogs/machine-learning/train-and-host-scikit-learn-models-in-amazon-sagemaker-by-building-a-scikit-docker-container/

An SVD recommender system can be trained with Scikit-Surprise library using a grid search to find the hyper-parameters that minimise the RMSE metric, as follows:

```
# Process ratings with Surprise Scikit
reader = Reader(line_format='user item rating timestamp', sep=',', rating_scale = (0.5,5
ratings = Dataset.load_from_file(ratings_path, reader=reader)
trainset = ratings.build_full_trainset()

# Find the best hyperparameters for an SVD model via grid search cross-validation
param_grid = {
    'n_factors': [10, 100, 500],
    'n_epochs': [5, 20, 50],
    'lr_all': [0.001, 0.005, 0.02],
    'reg_all': [0.005, 0.02, 0.1]}

gs_model = GridSearchCV(
    algo_class = SVD,
    param_grid = param_grid,
    n_jobs = -1,
    joblib_verbose = 5)

gs_model.fit(ratings)

# Train the SVD model with the parameters that minimise the root mean squared error
best_SVD = gs_model.best_estimator['rmse']
best_SVD.fit(trainset)
```

train.py hosted with ❤️ by GitHub view raw

The way to pack your algorithm is by using Docker containers that you can then push into AWS. Below you can see the Dockerfile I used for my model:

```
# Build an image that can do training and inference in SageMaker
# This is a Python 3 image that uses the nginx, gunicorn, flask stack
# for serving inferences in a stable way.

FROM ubuntu:18.04

MAINTAINER Amazon AI <sage-learner@amazon.com>

RUN apt-get -y update && apt-get install -y --no-install-recommends \
         wget \
         python3-pip \
         python3-setuptools \
         nginx \
         ca-certificates \
         gcc \
         libpq-dev \
         python3-dev \
         python3-venv \
         python3-wheel \
    && rm -rf /var/lib/apt/lists/*

RUN ln -s /usr/bin/python3 /usr/bin/python
RUN ln -s /usr/bin/pip3 /usr/bin/pip

# surprise won't install without numpy preinstalled
RUN pip3 install wheel

# Here we get all python packages.
# There's substantial overlap between scipy and numpy that we eliminate by
# linking them together. Likewise, pip leaves the install caches populated which uses
# a significant amount of space. These optimizations save a fair amount of space in the
# image, which reduces start up time.
RUN pip --no-cache-dir install numpy==1.16.2 flask gunicorn

# surprise won't install without numpy preinstalled
RUN pip install scikit-surprise

# Set some environment variables. PYTHONUNBUFFERED keeps Python from buffering our stand
# output stream, which means that logs can be delivered to the user quickly. PYTHONDONTW
# keeps Python from writing the .pyc files which are unnecessary in this case. We also u
# PATH so that the train and serve programs are found when the container is invoked.

ENV PYTHONUNBUFFERED=TRUE
ENV PYTHONDONTWRITEBYTECODE=TRUE
ENV PATH="/opt/program:${PATH}"
```

```
ENV PATH="/opt/program:${PATH}"

# Set up the program in the image
COPY src /opt/program
WORKDIR /opt/program
```

Dockerfile hosted with ❤️ by GitHub view raw

Once we have the Dockerfile ready we can build the image and push it into Amazon Elastic Container Registry (ECR). You can automate this process with the following script that is provided by AWS to automatically connect with your username and region, build the Docker image and make it available for SageMaker:

```
#!/usr/bin/env bash

# This script shows how to build the Docker image and push it to ECR to be ready for use
# by SageMaker.

# The argument to this script is the image name. This will be used as the image on the l
# machine and combined with the account and region to form the repository name for ECR.
image=$1

if [ "$image" == "" ]
then
    echo "Usage: $0 <image-name>"
    exit 1
fi

chmod +x src/train
chmod +x src/serve

# Get the account number associated with the current IAM credentials
account=$(aws sts get-caller-identity --query Account --output text)

if [ $? -ne 0 ]
then
    exit 255
fi


# Get the region defined in the current configuration (default to us-west-2 if none defi
region=$(aws configure get region)
region=${region:-us-west-2}


fullname="${account}.dkr.ecr.${region}.amazonaws.com/${image}:latest"

# If the repository doesn't exist in ECR, create it.

aws ecr describe-repositories --repository-names "${image}" > /dev/null 2>&1

if [ $? -ne 0 ]
then
    aws ecr create-repository --repository-name "${image}" > /dev/null
fi

# Get the login command from ECR and execute it directly
aws ecr get-login-password --region "${region}" | docker login --username AWS --passwordc
```

```
46
47  # Build the docker image locally with the image name and then push it to ECR
48  # with the full name.
49
50  docker build  -t ${image} .
51  docker tag ${image} ${fullname}
52
53  docker push ${fullname}
```

build_and_push.sh hosted with ❤️ by GitHub view raw

Once you run this script you can go to ECR and verify your image has been loaded in the registry.

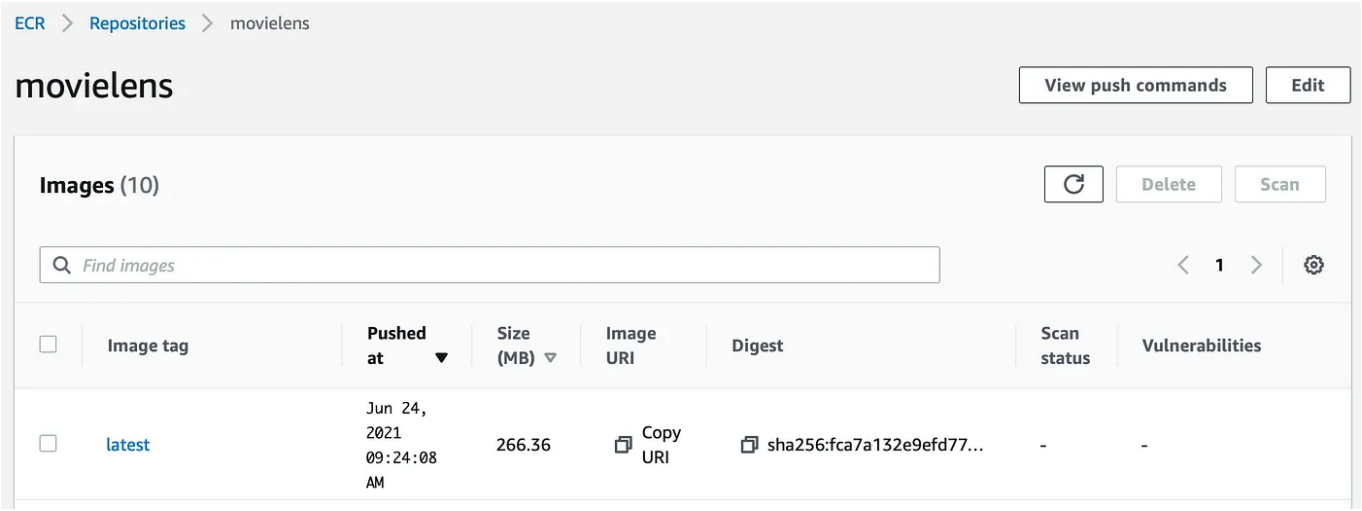


Image by Author

Training the model

AWS provides a great tool for building machine learning models, and this is SageMaker. This tool will run our Docker container, first for training the model with the movie ratings dataset and then, when our API is invoked, it will make rating inferences with the model, given a user and a movie IDs.

Creating a training job with SageMaker is very simple. We start by pointing to the path of our custom container in ECR. Having our own algorithm helps us to find the best model hyperparameters via a grid search in the code.

However, SageMaker also gives you the choice to use specific values in the training job configuration. The next step is to provide the channels which basically means configuring the input data sources, making sure you are using the same channel name in the AWS training job as in the model algorithm. Finally we just need to specify the output directory where the model will be saved.

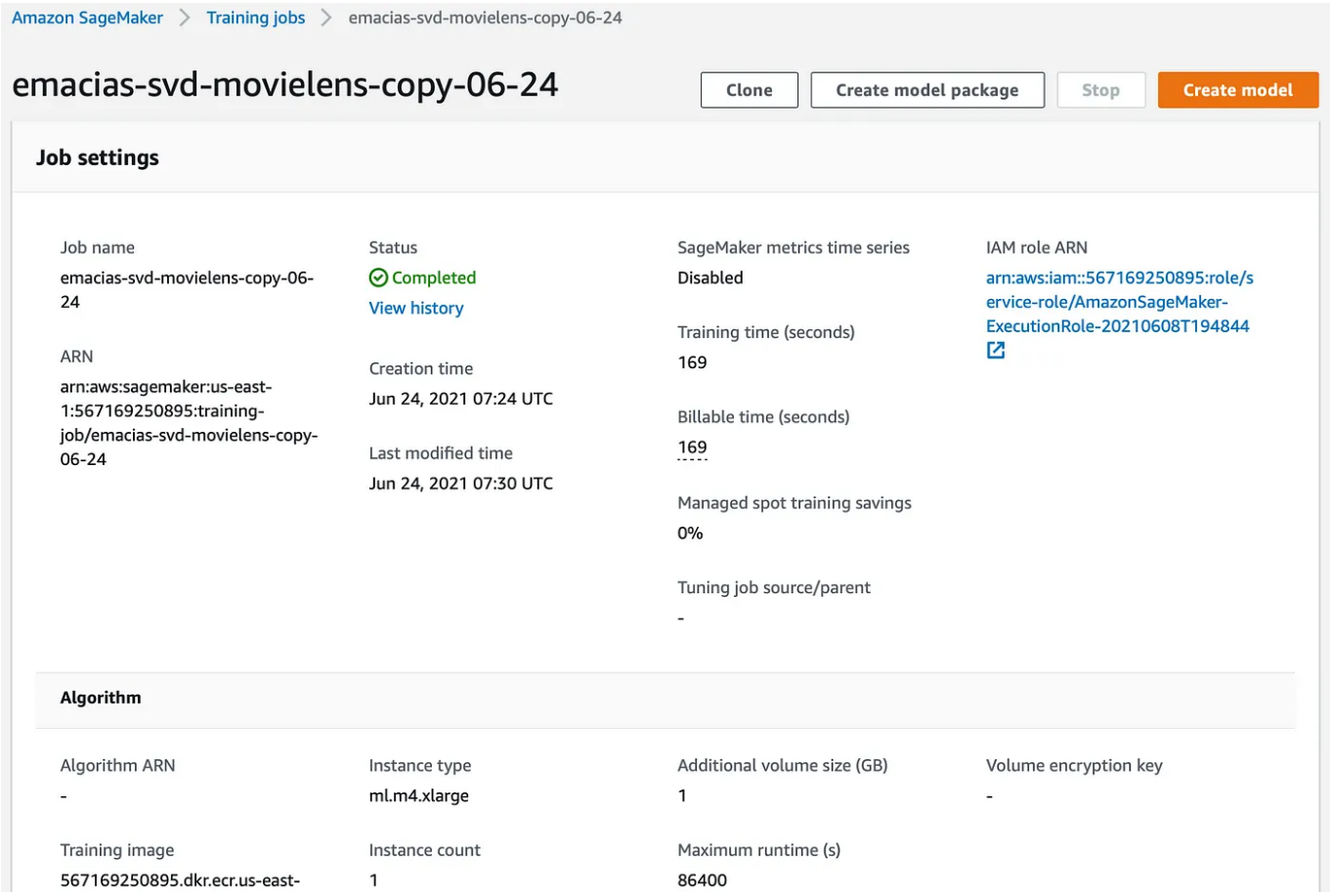


Image by Author

Once the training job is complete you can proceed to create an inference model from the job. Remember it is always a good practice to include some logs in your algorithm for debugging the different stages. The training logs can be found under the AWS CloudWatch service.

Great! Our movie recommender model is ready now. So how can we deploy our service?

Deploying the model

We'll keep on using the SageMaker service since we'll be creating a SageMaker Endpoint directly from our model. An endpoint configuration is required so you'll need to either choose an existing configuration or create a new one. What you need to be aware of in order to avoid a surprise in your bill is that you'll be charged for the time your SageMaker Endpoint is up, so make sure you remember to delete the endpoint when you no longer need it.

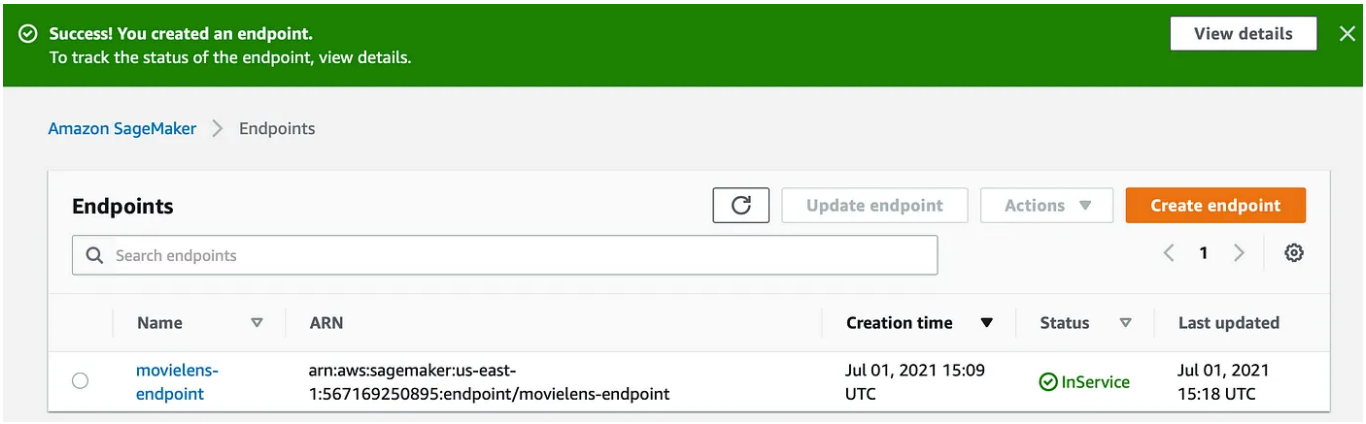


Image by Author

Once the endpoint is in service (it might take some time to launch), the next step in the deployment is to create a Lambda function that will invoke your endpoint, passing in a user and a movie, and expecting to receive a predicted rating. The way we link this function to our endpoint is by creating an environment variable whose value is the name of our SageMaker Endpoint and then loading this variable in the Lambda code.

One of the key benefits of using a Lambda function is that we can enrich the HTTP response returned by the SageMaker Endpoint. In our scenario, apart from returning the predicted movie rating, we'll also transform the movie id into the corresponding title. We can achieve this by making the Lambda function load a movie dataset (CSV file) that relates the movie IDs to titles. One thing to remember though is that any required Python library apart from the built-in ones (os, boto3, json) will have to be added as a Lambda layer. You can use the following lambda handler as a starting point:

```
'''
This code should be copied into a Lambda function of AWS.
Make sure to create an 'ENDPOINT_NAME' variable to points to your SageMaker Endpoint.
'''

import os
import boto3
import json
import pandas as pd

# grab environment variables
ENDPOINT_NAME = os.environ['ENDPOINT_NAME']

runtime = boto3.client('runtime.sagemaker')
s3_client = boto3.client('s3')
bucket = <YOUR_BUCKET_NAME>
movies_filename = 'movies.csv'

def lambda_handler(event, context):

    payload = json.dumps(event)

    response = runtime.invoke_endpoint(EndpointName=ENDPOINT_NAME,
                                       ContentType='application/json',
                                       Body=payload)

    file_contents = s3_client.get_object(Bucket=bucket, Key=movies_filename)
    movies = pd.read_csv(file_contents['Body'], usecols=[0,1], index_col=0, squeeze=True

    movie_id = event['movieId']
    predicted_value = movies[int(movie_id)] + ' : ' + response['Body'].read().decode()

    return predicted_value
```

lambda_function.py hosted with ❤️ by GitHub view raw

Before moving on to the next step, you should make sure your Endpoint and Lambda function are working correctly. Lambda service provides an easy tool to test your function, all you need to do is configuring a test event where you pass in a sample input to verify the correct result of the query. In our

```
{
 “userId”: “2”,
 “movieId”: “2959”
}
```

After running the test, and once the Endpoint response is enriched by the Lambda function, we’ll obtain the following response:

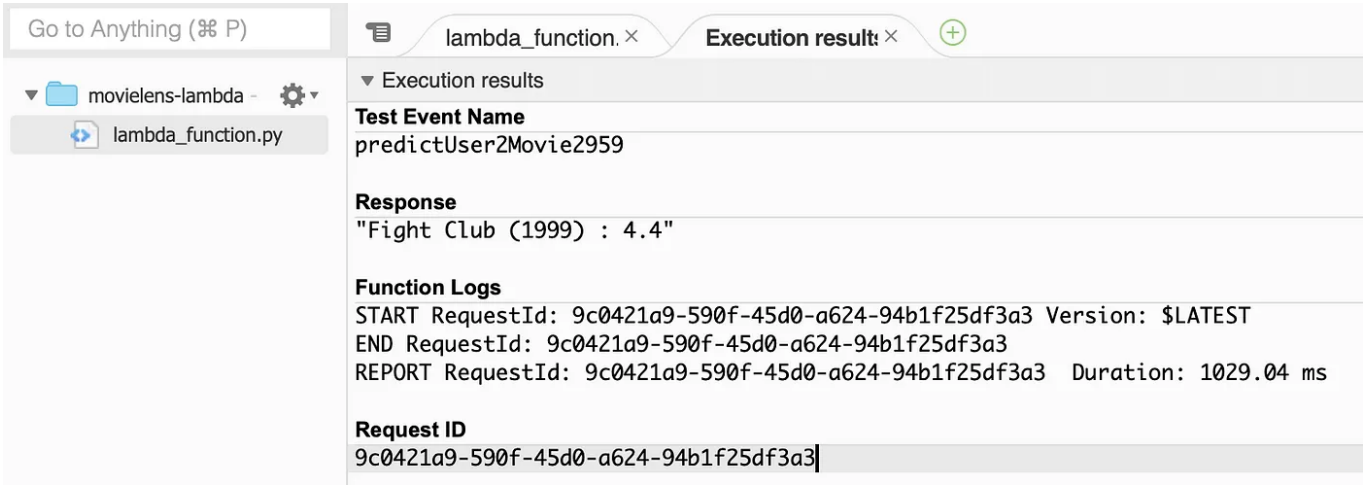


Image by Author

The last step of our ML deployment is to create a public API that can be invoked from outside our AWS private virtual cloud. API Gateway is the service that will let us build a public REST API from scratch. All we need to do inside this service is creating a resource and a method (in our case, POST, since we’ll need to provide the user ID and movie ID in the body of the request), making sure to choose the option to integrate this REST method with our Lambda function.

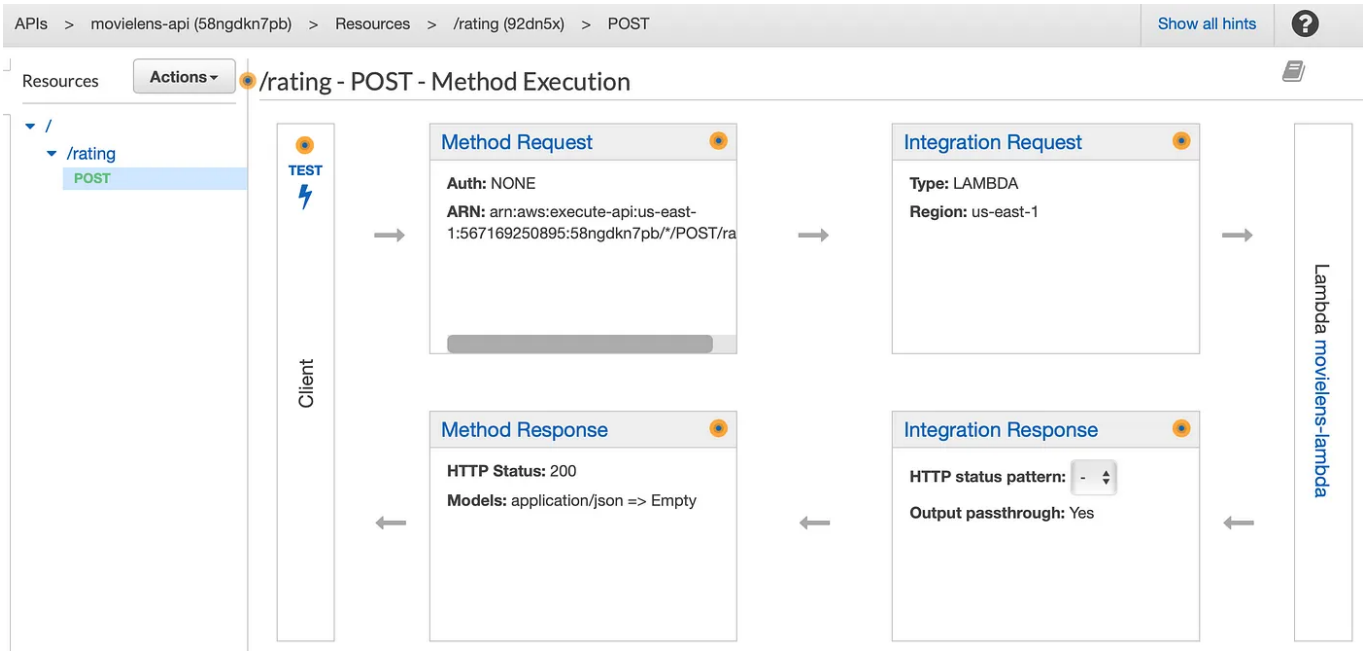


Image by Author

Our recommender system is now live! API Gateway provides the public URL we need to invoke our service. It's time to ask for some movie predictions...

Testing our AI service

There are different ways you can invoke our movie recommender. If you prefer to do it via the terminal, you can use a command-line utility such as *curl*. Alternatively you can use a visual tool such as Postman, as we're doing here. Just remember we have configured our service with a POST method, so in Postman, we'll have to choose this type of method, enter the URL we got from API Gateway and include a JSON body in the HTTP request. The expected format of the JSON body is as follows:

```
{
 "userId": String,
```

```
  “movieId”: String
}
```

And voilà! Below we can see the response from our API including the name of the movie (*Fight Club*) and the predicted rating from the user (*4.4*).

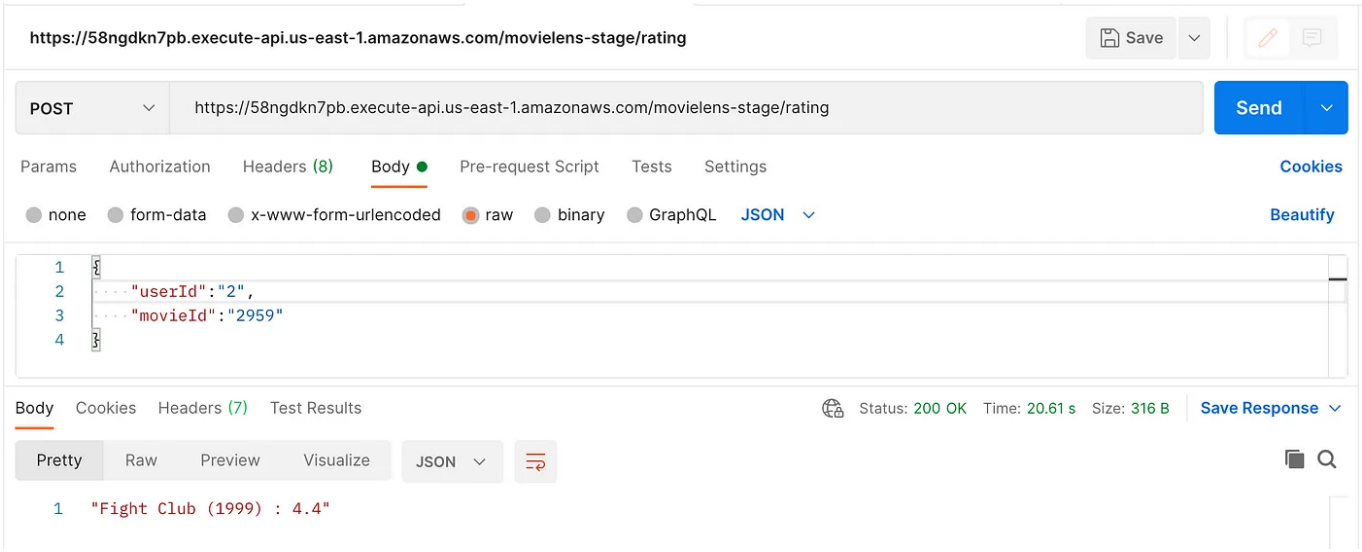


Image by Author

Conclusion

We have covered how we can take a custom machine learning algorithm into production via a public cloud provider such as AWS, applying it to the use case of a recommender system that can predict movie ratings from the users.

A similar outcome can be achieved with Google Cloud Platform or Microsoft Azure, so it is up to you to pick the public cloud provider that meets your requirements in the most efficient and economic way.

Feel free to use my code for your own projects or for learning purposes: https://github.com/ejmacias/aws_movie_recommender

Happy deployment!

Recommender Systems AWS Sagemaker Flask Docker

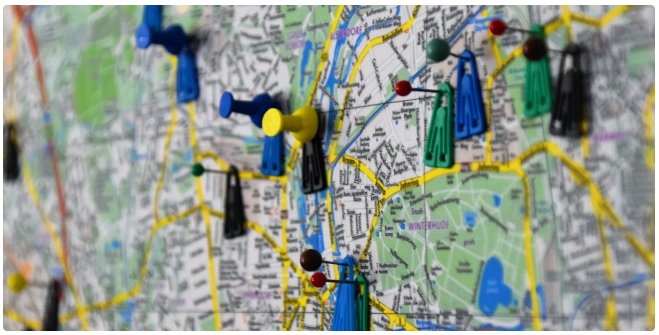
Written by Emilio Macias

Follow

34 Followers

Software engineer with a passion for data science and its social impact.

More from Emilio Macias



Emilio Macias in Towards Data Science

Store locations

Emilio Macias in Towards Data Science

Clustering news articles based on named entities

Finding the optimal location for a new low-cost supermarket in Madrid

8 min read · Dec 22, 2020

32 2

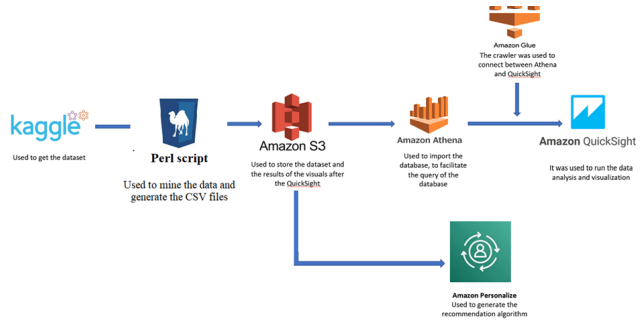
How to group real-time news in Spanish by people, locations and organisations

12 min read · May 2, 2021

7 1

See all from Emilio Macias

Recommended from Medium



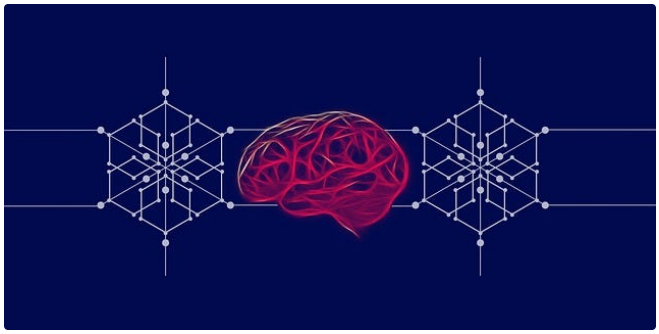
Sahar Farhat

Recommendation system for e-commerce website

I. Scenario Definition

5 min read · Oct 13

1



ai geek (wishesh)

Best Practices for Deploying Large Language Models (LLMs) in...

Large Language Models (LLMs) have revolutionized the field of natural language...

10 min read · Jun 26

209 1

Lists

Coding & Development

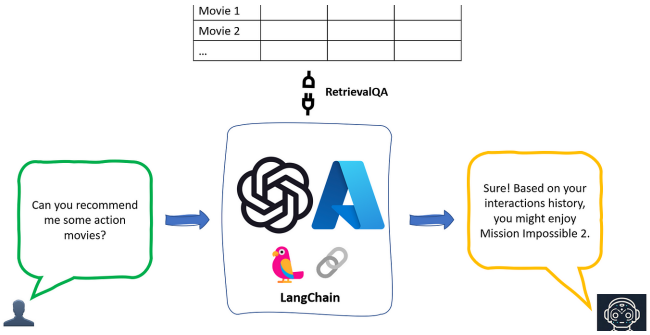
11 stories · 258 saves

New_Reading_List

174 stories · 180 saves

Natural Language Processing

803 stories · 370 saves



Valentina Alto in Microsoft Azure

Enhancing Recommendation Systems with Large Language...

An implementation with LangChain and Azure OpenAI

✨ · 13 min read · Aug 23

161

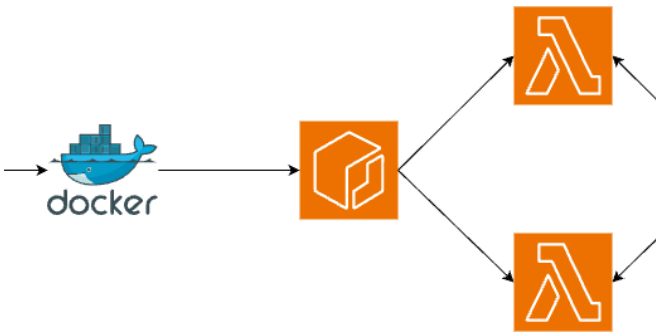
Amy @GrabNGoInfo in GrabNGoInfo

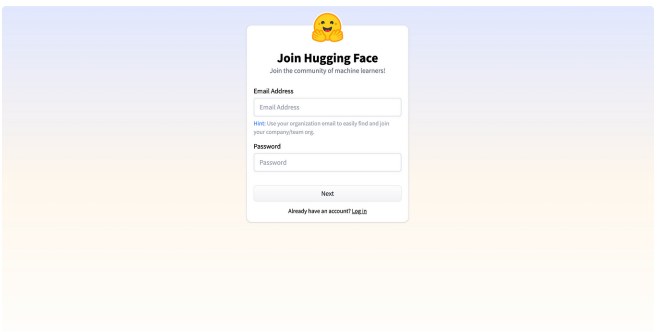
Hybrid Recommendation System Using User-based and Item-base...

4 hybrid methods to combine user-user and item-item collaborative filtering

✨ · 8 min read · Jul 11

11





FanchenBao

sriram c in Technology at Nineleaps

API Service with FastAPI + AWS Lambda + API Gateway and Make ...

The goal of this article is to show one way to create a FastAPI app, deploy it to AWS...

14 min read · Sep 12

36

How to deploy your own LLM(Large Language Models)

If you are new to LLM world, Then I would suggest you to go through previous articles t...

5 min read · Sep 17

11

See more recommendations