

MSIN0166: Data Engineering

Individual Coursework

Symeon Kokovidis

18116815

29th April 2020

Table of contents

Definitions	3
Introduction.....	4
The dataset of this project.....	4
Servers' setup.....	5
Internal API in Faculty.ai.....	5
External API in a docker image	7
External API with Azure Machine Learning Studio.....	14
Client's call.....	17
Suggestions for further Improvement.....	17
References	18

Definitions

Services

Faculty.ai

A Data Science cloud environment which offers compute resources with easy configuration collaboration and deployment. Faculty.ai allow users to share their code with other team members mainly with Jupyter Notebooks which can run Python or R. It offers unlimited file-size upload, automated data exploration and scalable computing power. Users can also open a terminal if they want to develop more complex set-ups. Based on personal experience, Faculty.ai is a similar platform to Databricks but focuses mostly on single-node computing

Among others, Faculty.ai offers a native user interface with MLflow Python package and API deployment capabilities with Flask and other Python packages (see package's definitions below for further information).

Python Packages

MLflow

According to its documentation, MLflow is an open source platform to manage the Machine Learning lifecycle, including experimentation, reproducibility, deployment, and a central model registry (namely: MLflow Tracking, MLflow Projects, MLflow Models, MLflow Model Registry). In our project we exploit the MLflow Tracking tool which will help us determine the best performing Machine Learning model in terms of accuracy and training/predicting time.

MLflow has built-in integrations, with several services and packages that we use in this project. Namely, it has integrations with Apache Spark, Scikit-learn, Docker, Amazon SageMaker, Azure Machine Learning and Google Cloud.

Flask

Flask is a micro web framework written in Python. The easy setup allows to create entry-level APIs in Python. These APIs can be used from internal clients or can be deployed on the web. It offers debugging and unit testing capabilities, RESTful request dispatching, extensive documentation and compatibility with popular cloud services (for example Google App Engine).

Software

Postman for Mac OSX

Postman is an API client that is used from developers to create, test, document and share APIs. The client allow users to create HTTP/s requests and read their responses.

In our project we have used Postman to make queries with GET and POST methods to our APIs.

Introduction

For this project, we have created three APIs that provide movie recommendations. The first API was developed in Faculty.ai with intention to be used from internal clients. For selecting our model we used the surprise package with the use of MLflow. After the experimentation with different algorithms we used for our final model the Singular Value Decomposition (SVD) algorithm. The second API was developed in a Google Cloud Virtual Machine and has used the Alternating Least Square (ALS) Matrix Factorization algorithm. After the successful run of the API, we pushed our environment to a repo on Docker Hub. For checking that our docker image works properly, we used the same container on the Amazon Web Services. Finally, for our third API we exploited the Azure Machine Learning Studio and the Matchbox Algorithm which was developed by Microsoft. With these three APIs we fetched movie recommendations for our test-users. For these user-movie pairs we requested scores from a SageMaker API that was developed for the needs of this project. Finally, based on the best scores, we provided five movie recommendations to our test-users.

All of the APIs were used for Collaborative Filtering. Collaborative Filtering actually refers to recommendations based on the activity of other similar users to the one the we examine.

The repository of the project can be found on our personal directory on Faculty.ai and on GitHub: <https://github.com/uceisko/MSIN0166-Individual>

The dataset of this project

In this project we have used the MovieLens 100K Dataset from GroupLens Research Lab. GroupLens is a research lab in the Department of Computer Science and Engineering at the University of Minnesota. MovieLens 100K Dataset is used for benchmark from many popular recommendation algorithms.

The data collection consists of 100.000 ratings (in scale of 1 to 5) from 943 users on 1682 movies. According to official documentation the data was collected through the MovieLens web site (movielens.umn.edu) during the seven-month period from September 19th, 1997 through April 22nd, 1998. The dataset has information for users with at least 20 ratings.

In our project we have used the following files from the data collection:

- u.data: The full dataset. Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of id | item id | rating | timestamp. Below you will find a preview of it:

userId	movieId	rating	timestamp
0	1	1	5 874965758
1	1	2	3 876893171
2	1	3	4 878542960
3	1	4	3 876893119
4	1	5	3 889751712
...
99995	943	1067	2 875501756
99996	943	1074	4 888640250
99997	943	1188	3 888640250
99998	943	1228	3 888640275
99999	943	1330	3 888692465

100000 rows x 4 columns

The u.data dataset

- u.item: Which contains Information about the items (movies); this is a tab separated list of : movie id | movie title | release date | video release date | IMDb URL | unknown | Action | Adventure | Animation | Children's | Comedy | Crime | Documentary | Drama | Fantasy | Film-Noir | Horror | Musical | Mystery | Romance | Sci-Fi | Thriller | War | Western. The movie ids are the ones used in the u.data data set. In our project we have used only the movie id and the movie title attribute. Below you will find a preview of it:

movieid		title
0	1	Toy Story (1995)
1	2	GoldenEye (1995)
2	3	Four Rooms (1995)
3	4	Get Shorty (1995)
4	5	Copycat (1995)
...
1677	1678	Mat' i syn (1997)
1678	1679	B. Monkey (1998)
1679	1680	Sliding Doors (1998)
1680	1681	You So Crazy (1994)
1681	1682	Scream of Stone (Schrei aus Stein) (1991)

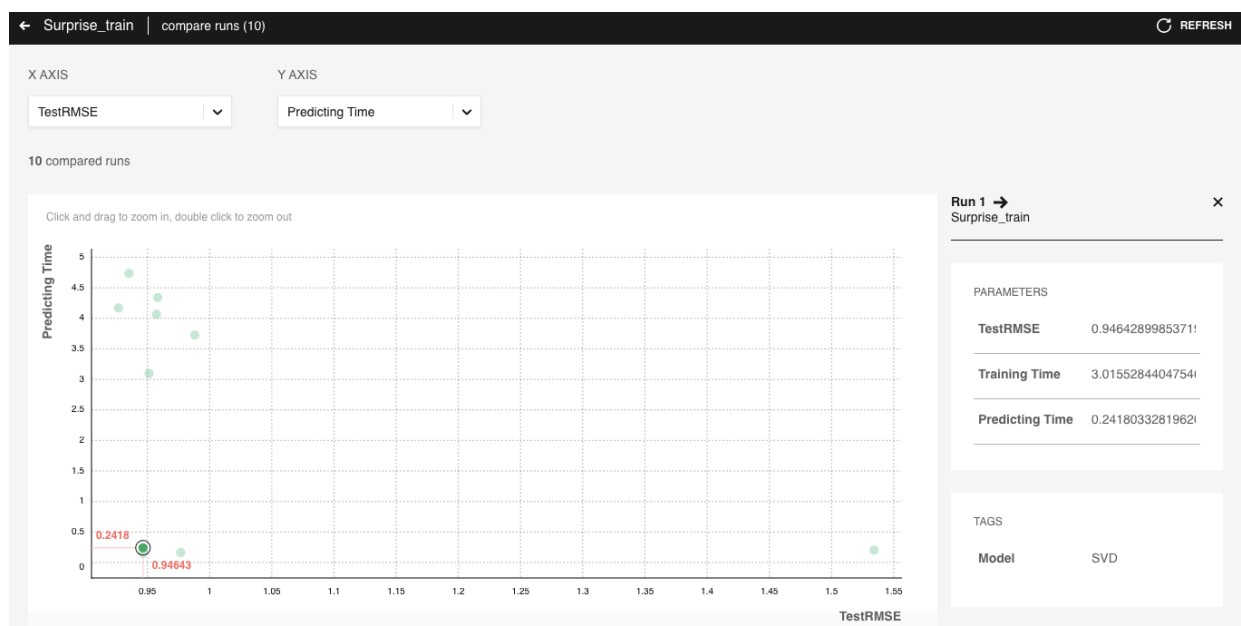
1682 rows × 2 columns

The u.item dataset with only
the movie ID and title attributes

Servers' setup

Internal API in Faculty.ai

For the selection of the model for our first API, we have used the surprise recommender systems library with MLflow. We examined 10 different recommender algorithms and we logged their performance with MLflow. MLflow actually stored the training/predicting time and their accuracy (Root Mean Square Error). With Faculty's native MLflow UI, we selected the best performing algorithm not only in terms of predictive accuracy but also in terms of predicting time.



The best algorithm was the Singular Value Decomposition (SVD).

For this algorithm, we performed a Grid Cross Validated hyper-parameter search and we scheduled a job on Faculty to update the best performing hyper-parameters once per day. With the best performing hyper-parameters we deployed an API with Flask and Faculty's deployment services.

The screenshot shows the Faculty deployment interface for an application named 'surprise_app'. At the top, there are tabs for 'Configure', 'Test', and 'Deploy', with 'Deploy' being the active tab. Below the tabs, there's a 'DEPLOYMENT STATUS' section showing 'DEPLOYED' in green. To the right of this status are buttons for 'REDEPLOY' and 'STOP'. Below the status, it shows 'Last deployed at: 28 April 2020, 15:53' and 'Last deployed by: symeon-kokovidis'. To the right of this, there's a 'Server:' section showing a green 'RUNNING' status, '1 Core', and '4 GB' of memory. Below the server status, there are two tabs: 'API Logs' and 'Environment Logs', with 'API Logs' being the active tab. The logs show a series of messages from the gunicorn server, including starting the server, listening at http://127.0.0.1:8000, and booting workers. The final log entry shows a successful GET request to /predict/11 with a status of 200 and a response size of 359 bytes.

Finally, with Postman app we tested if the API works properly:

The screenshot shows the Postman interface for an 'Untitled Request'. The request method is 'GET' and the URL is 'https://svdsurprise.api.ucl.my.faculty.ai/predict/11'. The 'Headers' tab is selected, showing a table with one header: 'UserAPI-Key' with the value 'c07884p7ba32k6osq57jfr01q9if18rgi04v1ir51e:...' and a description 'Key'. The 'Body' tab is also selected, showing the response in JSON format. The response status is '200 OK', the time is '628 ms', and the size is '580 B'. The response body is a JSON object with a 'movieId' field containing an array of 10 movie IDs and a 'pred_rating' field containing an array of 10 predicted ratings.

KEY	VALUE	DESCRIPTION
UserAPI-Key	c07884p7ba32k6osq57jfr01q9if18rgi04v1ir51e:...	Key

```
{
  "movieId": {
    "0": 513,
    "1": 408,
    "2": 189,
    "3": 169,
    "4": 963,
    "5": 483,
    "6": 19,
    "7": 515,
    "8": 474,
    "9": 498
  },
  "pred_rating": {
    "0": 4.5150486475,
    "1": 4.5019434087,

```

The deployed API was finally used on the client's scenario.

External API in a docker image

For our next API, we developed in Apache Spark (with Pyspark package) a recommender which works with unseen users. The concept was to create a POST API that take new users, with no prior ratings, retrain an Alternating Least Square (ALS) algorithm and return movie recommendations for them.

Moreover, we hosted the API in a Google Cloud Platform virtual machine. Once we have ensured that the API works properly, we dockerized the app and hosted it on Docker Hub. To ensure that our repo is available for reproduction, we pulled it and deployed it in an AWS EC2 machine.

In the directory 1. ServersSetup → 1.2 DockerApi of the GitHub repo there are three python script files for this task.

- The `initialize_data.py` actually removes the test users (and their ratings) that we are going to get recommendations for them in the Client's setup chapter.
- The `cross_validation.py` performs a cross-validated grid search to get the best hyperparameters for our ALS algorithms. These hyperparameters are saved in a .json file
- The `app.py` actually holds our Flask server and our POST API. In more detail, the script loads the existing ratings , parses a new user's ratings, retrains the ALS algorithm (with the best hyperparameters from `cross_validation.py`) and returns movie recommendations for the user with a .json file. These new -unseen- ratings are saved to our local dataset for further improvement of our model.

The accompanied '1.2 Dev: Docker API' Jupyter Notebook on the same directory gives a more insightful view for these three scripts.

The initial deployment of the app was done in a Virtual Machine on Google Cloud Platform. We have selected a region which will be close to our potential clients

The screenshot shows the Google Cloud Platform VM configuration interface. The VM name is 'als-recommender'. The region is 'europe-west6 (Zürich)' and the zone is 'europe-west6-a'. The machine configuration is set to 'General-purpose' family, 'N1' series, and 'n1-standard-4' machine type (4 vCPU, 15 GB memory). The estimated monthly cost is \$136.86.

Name ⓘ
Name is permanent
als-recommender

Labels ⓘ (Optional)
+ Add label

Region ⓘ
Region is permanent
europe-west6 (Zürich)

Zone ⓘ
Zone is permanent
europe-west6-a

Machine configuration

Machine family
General-purpose
Machine types for common workloads, optimized for cost and flexibility

Series
N1
Powered by Intel Skylake CPU platform or one of its predecessors

Machine type
n1-standard-4 (4 vCPU, 15 GB memory)

Summary

	vCPU	Memory
	4	15 GB

Cost

\$136.86 monthly estimate
That's about \$0.187 hourly
Pay for what you use: No upfront costs and per second billing
[Details](#)

For the Operating System we used the Ubuntu 18.04 LTS Minimal version:

Public images

Custom images

Snapshots

Existing disks

Operating system

Ubuntu

Version

Ubuntu 18.04 LTS Minimal

amd64 bionic minimal image built on 2020-04-15, supports Shielded VM features

Boot disk type

Standard persistent disk

Size (GB)

20

We requested full access to all cloud APIs and we allows HTTP/HTTPS traffic:

Identity and API access

Service account

Compute Engine default service account

Access scopes

☐ Allow default access

☒ Allow full access to all Cloud APIs

☐ Set access for each API

Firewall

Add tags and firewall rules to allow specific network traffic from the Internet

☒ Allow HTTP traffic

☒ Allow HTTPS traffic

Once we have created the Virtual Machine, we opened an incoming port for our API. To achieve this, we search on the top search of the project for the Firewall Rules. We created a new rule and opened the incoming tcp:8080 port :

Target tags *

http-server

Source filter

IP ranges

Source IP ranges *

Second source filter

None

Protocols and ports

☐ Allow all

☒ Specified protocols and ports

☒ tcp : 8080

☐ udp : all

☐ Other protocols

protocols, comma separated, e.g. ah, sctp

Finally, we connected with the web-based SSH client to the instance:

VM instances

[CREATE INSTANCE](#)
[IMPORT VM](#)
[REFRESH](#)
[START](#)
[STOP](#)
[RESTART](#)
[DELETE](#)
[SHOW INFO PANEL](#)

Filter VM instances

Columns

<input type="checkbox"/>	Name	Zone	Recommendation	In use by	Internal IP	External IP	Connect
<input checked="" type="checkbox"/>	als-recommender	europe-west6-a			10.172.0.5 (nic0)	34.65.252.26	SSH
<input type="checkbox"/>	als-flask	europe-west6-a			10.172.0.4 (nic0)	None	SSH

Inside the Virtual Machine we performed the following steps:

1. Installed git:

```
sudo apt update
sudo apt install git
```

2. Copied our GitHub repo:

```
git clone
https://uceisko:\[password\]@github.com/uceisko/MSIN0166-Individual
```

3. Once we have successfully pulled our repo we navigated to the working directory of the API:

```
cd MSIN0166-Individual/1.\ ServersSetup/1.2\ DockerAPI/
```

4. Install all the required packages for using Flask and Pyspark:

```
sudo bash ./'A. install.sh'
```

5. Executed the three script files

```
python 'B. initialize_data.py'
python 'C. cross_validation.py'
python 'D. app.py'
```

The last script actually deploys the Flask API. In a successful run we will have the following message from Flask:

```
uceisko@als-recommender:~$ cd MSIN0166-Individual/1.\ ServersSetup/1.2\ DockerAPI/
uceisko@als-recommender:~/MSIN0166-Individual/1. ServersSetup/1.2 DockerAPI$ python 'D. app.py'
20/04/28 21:18:42 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-ja
va classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
* Serving Flask app "D. app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
* Restarting with stat
20/04/28 21:18:51 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-ja
va classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/04/28 21:18:52 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
* Debugger is active!
* Debugger PIN: 189-680-382
```

POST

http://34.65.252.26:8080

Send

Save

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

```

1 {
2   "userId": {
3     "1259": 11,
4     "1260": 11,
5     "1261": 11,
6     "1262": 11,
7     "1263": 11,
8     "1264": 11,
9     "1265": 11,
10    "1266": 11
  }
}

```

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 5.36 s

Size: 1.43 KB

Save Response

Pretty

Raw

Preview

Visualize

JSON

```

1 [{"userId":{"0":11,"1":11,"2":11,"3":11,"4":11,"5":11,"6":11,"7":11,"8":11,"9":11,"10":11,"11":11,"12":11,"13":11,"14":11,"15":11,"16":11,"17":11,"18":11,"19":11,"title":{"0":"Sleepless in Seattle (1993)","1":"Shawshank Redemption, The (1994)","2":"Searching for Bobby Fischer (1993)","3":"Four Weddings and a Funeral (1994)","4":"Hoop Dreams (1994)","5":"Braveheart (1995)","6":"Three Colors: Blue (1993)","7":"Apollo 13 (1995)","8":"Godfather, The (1972)","9":"Taxi Driver (1976)","10":"Madness of King George, The (1994)","11":"Jurassic Park (1993)","12":"Three Colors: White (1994)","13":"Crimson Tide (1995)","14":"Quiz Show (1994)","15":"Postino, Il (1994)","16":"Star Wars (1977)","17":"Truth About Cats & Dogs, The (1996)","18":"Maverick (1994)","19":"Fugitive, The (1993)"},"prediction":{"0":2.9729223251,"1":2.9364533424,"2":2.8724303246,"3":2.8660275936,"4":2.863401413,"5":2.729903698,"6":2.6895344257,"7":2.6374571323,"8":2.6332607269,"9":2.602656126,"10":2.5806727409,"11":2.5510993004,"12":2.4941391945,"13":2.4758052826,"14":2.4546203613,"15":2.4450564384,"16":2.4450368881,"17":2.4439074993,"18":2.4396026134,"19":2.4300711155}}]

```

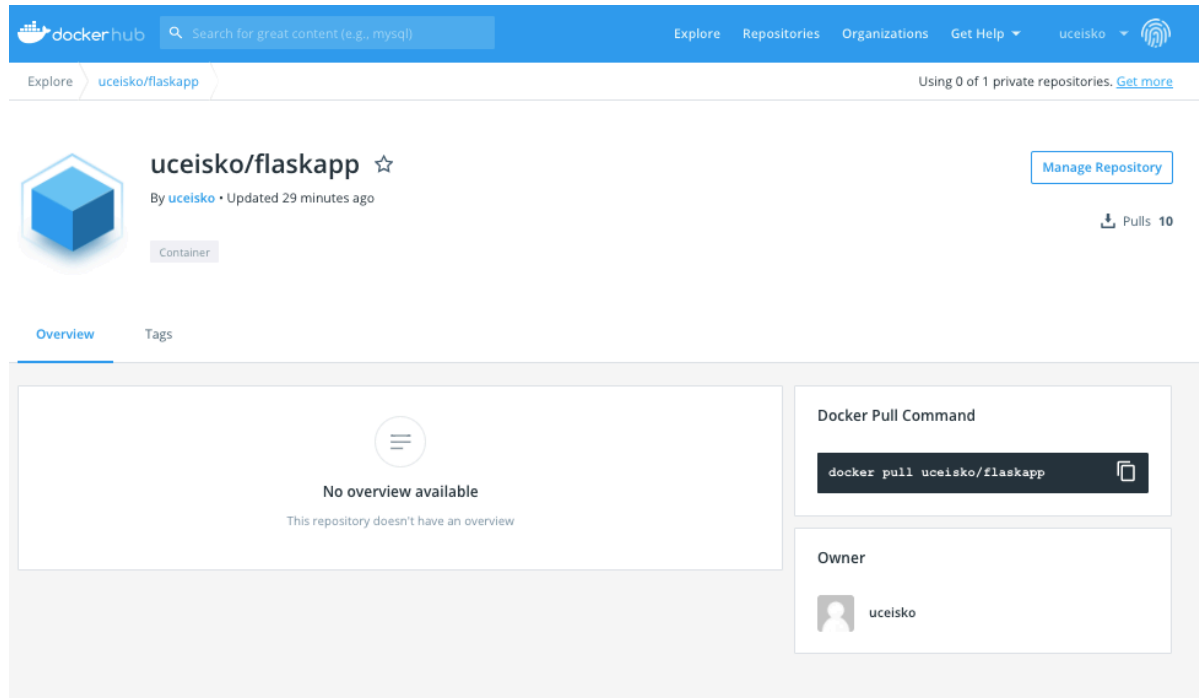
In the following steps we demonstrate how we dockerized our app and pushed it to Docker hub.

```
sudo bash ./docker_install.sh
```

```
sudo bash ./build_repo_docker_hub.sh
```

```
An image does not exist locally with the tag: uceisko/flaskapp
root@als-recommender:~/MSIN0166-Individual/1. ServersSetup/1.2 DockerAPI# docker tag uceisko_flask:latest uceisko/flaskapp:0.1
root@als-recommender:~/MSIN0166-Individual/1. ServersSetup/1.2 DockerAPI# docker push uceisko/flaskapp:0.1
The push refers to repository [docker.io/uceisko/flaskapp]
cc3fce039f43: Pushed
7dlbf7cf12fe: Pushing [=====>] 178.2MB/248.6MB
bce384ec3660: Pushed
e42fa59f390c: Pushed
72e75bd7f43a: Pushed
5863cbd5c294: Pushed
96e2bee8f963: Pushed
3a9711d73112: Pushing [=====>] 114.2MB/421.6MB
7150649695f0: Pushed
4e6122f633eb: Pushing [=====>] 59.53MB/381.4MB
590a9f21eb02: Pushing [=====>] 19.2MB/28.36MB
28ba7458d04b: Waiting
838a37a24627: Waiting
a6ebef4a95c3: Waiting
b7f7d2967507: Waiting
```

After the successful commit we should be able see our repo on Docker Hub:

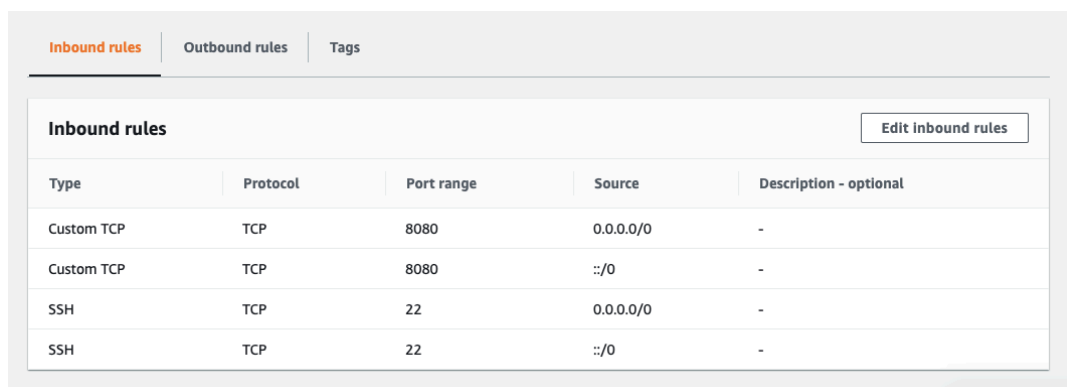


The screenshot shows the Docker Hub interface for the repository 'uceisko/flaskapp'. The page includes a search bar, navigation links (Explore, Repositories, Organizations, Get Help), and a user profile 'uceisko'. The repository page features a blue cube icon, the name 'uceisko/flaskapp', and a 'Container' label. A 'Manage Repository' button is visible. The 'Overview' tab is selected, showing a message: 'No overview available. This repository doesn't have an overview'. A 'Docker Pull Command' box displays the command 'docker pull uceisko/flaskapp'. The 'Owner' section shows the user 'uceisko'.

At the end of this stage, we closed our Google Cloud Virtual Machine instance to avoid further costs.

In the next steps we show how we pulled the repo to a new Virtual Machine on Amazon Web Services.

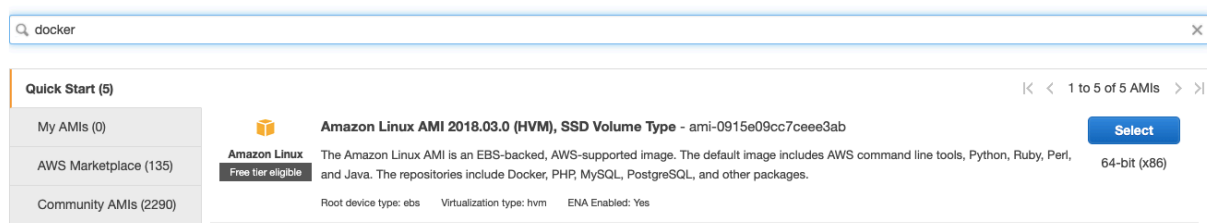
In the EC2 Services we accessed the security groups. We created a new security group which allows inbound traffic for the 8080 port:



The screenshot shows the AWS Management Console 'Inbound rules' tab for a security group. It displays a table of inbound rules:

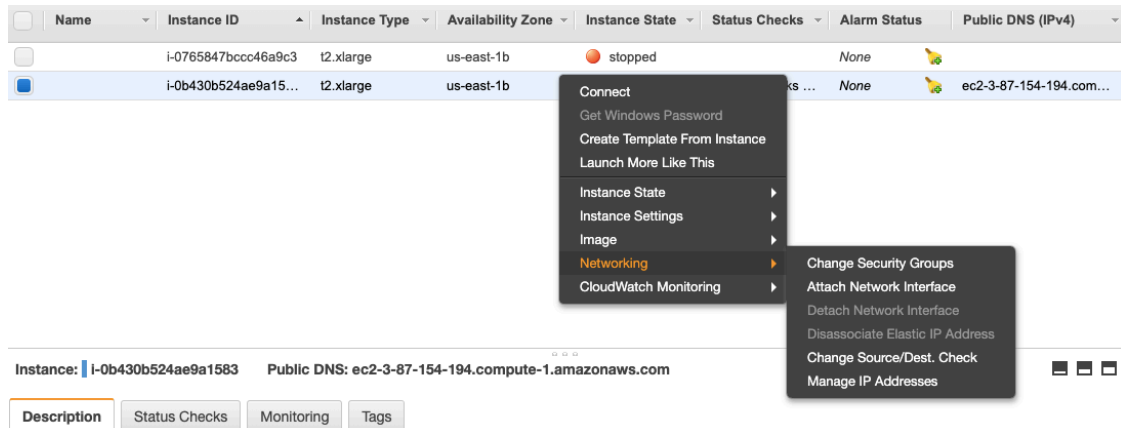
Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	8080	0.0.0.0/0	-
Custom TCP	TCP	8080	:::0	-
SSH	TCP	22	0.0.0.0/0	-
SSH	TCP	22	:::0	-

Then, we launched a new instance with docker pre-installed:

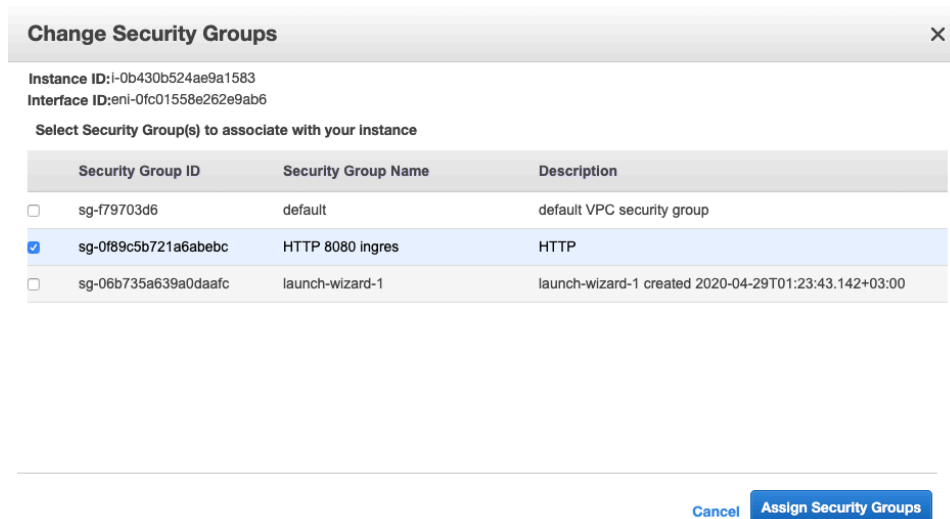


The screenshot shows the AWS 'Quick Start' page for selecting an Amazon Machine Image (AMI). The search results show the 'Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type' with the ID 'ami-0915e09cc7ceee3ab'. The page includes a 'Select' button and a '64-bit (x86)' label.

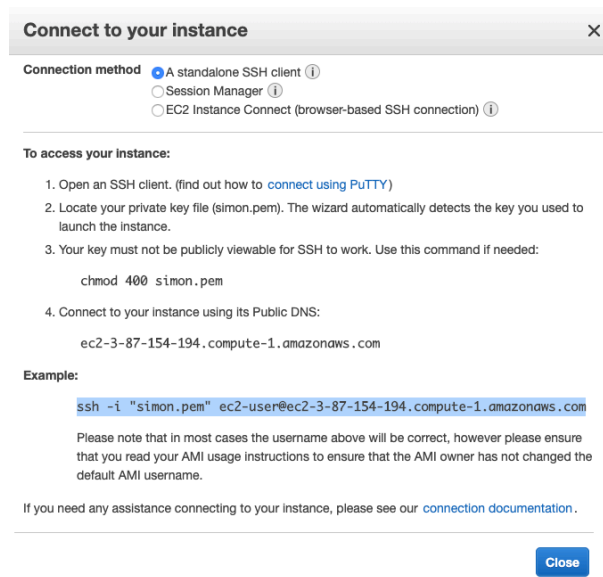
Before we access the instance we changed its security group to the one that we have created. We right clicked on the instance, and we access the Security Groups:



We selected our new security group which allows inbound requests to tcp:8080 port



With Mac OSX terminal we accessed the instance:



Finally, we pulled the image from our Docker Hub repo and we ran it:

```
sudo service docker restart
```

```
sudo service docker status
```

```
sudo docker run hello-world
```

```
sudo docker pull uceisko/flaskapp:0.1
```

```
sudo setfacl -m user:$USER:rw /var/run/docker.sock #get root  
access to the app
```

```
sudo docker images #get image id
```

```
docker run -d -p 8080:8080 a3a04a644b09 #use the image id as  
last argument
```

The commands can be found also on our working directory on GitHub (pull_repo.sh).
Once we have successfully ran the pulled image, we made queries to our API:

The screenshot shows a web browser interface for a REST client. The top bar indicates a POST request to `http://3.87.154.194:8080`. Below the bar, there are tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results'. The 'Body' tab is selected, showing a JSON response. The response is a JSON object with a 'userId' field containing an array of movie titles and their corresponding prediction values. The response status is 200 OK, time is 8.06 s, and size is 1.41 KB. The response is displayed in a 'Pretty' format, showing the JSON structure with indentation.

```
{  
  "userId": {  
    "1259": 11,  
    "1260": 11,  
    "1261": 11,  
    "1262": 11,  
    "1263": 11,  
    "1264": 11,  
    "1265": 11,  
    "1266": 11  
  }  
}
```

The 'Test Results' tab shows the following details: Status: 200 OK, Time: 8.06 s, Size: 1.41 KB. There is a 'Save Response' button.

The 'Body' tab shows the following JSON response:

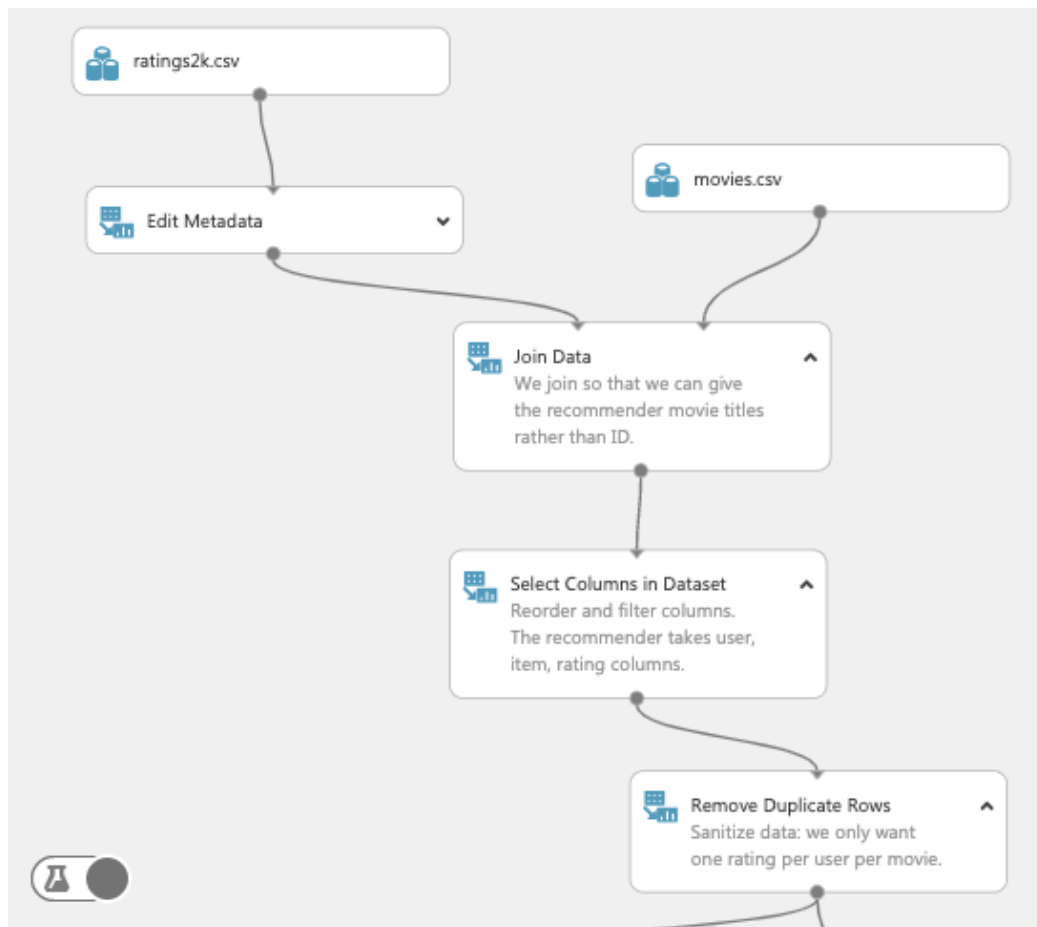
```
{  
  "userId": {  
    "0": "Braveheart (1995)",  
    "1": "Four Weddings and a Funeral (1994)",  
    "2": "Shawshank Redemption, The (1994)",  
    "3": "Star Wars (1977)",  
    "4": "Usual Suspects, The (1995)",  
    "5": "Godfather, The (1972)",  
    "6": "Dances with Wolves (1990)",  
    "7": "Silence of the Lambs, The (1991)",  
    "8": "Terminator 2: Judgment Day (1991)",  
    "9": "Fugitive, The (1993)",  
    "10": "Jurassic Park (1993)",  
    "11": "Dead Man Walking (1995)",  
    "12": "Toy Story (1995)",  
    "13": "Rock, The (1996)",  
    "14": "Apollo 13 (1995)",  
    "15": "Blade Runner (1982)",  
    "16": "Fargo (1996)",  
    "17": "Phenomenon (1996)",  
    "18": "Mr. Holland's Opus (1995)",  
    "19": "Forrest Gump (1994)",  
    "prediction": {  
      "0": 3.0085263252,  
      "1": 2.9363632202,  
      "2": 2.9182054996,  
      "3": 2.8913507462,  
      "4": 2.8698308468,  
      "5": 2.7051150799,  
      "6": 2.6847853661,  
      "7": 2.6819050312,  
      "8": 2.6180589199,  
      "9": 2.5877215862,  
      "10": 2.5874800682,  
      "11": 2.5296406746,  
      "12": 2.5262260437,  
      "13": 2.435092926,  
      "14": 2.4294247627,  
      "15": 2.394276619,  
      "16": 2.3569624424,  
      "17": 2.2950811386,  
      "18": 2.2857394218,  
      "19": 2.2751111984  
    }  
  }  
}
```

Note that in this request, we have changed the IP address to the one of our new instance.
This API will be used in our client's side scenario.

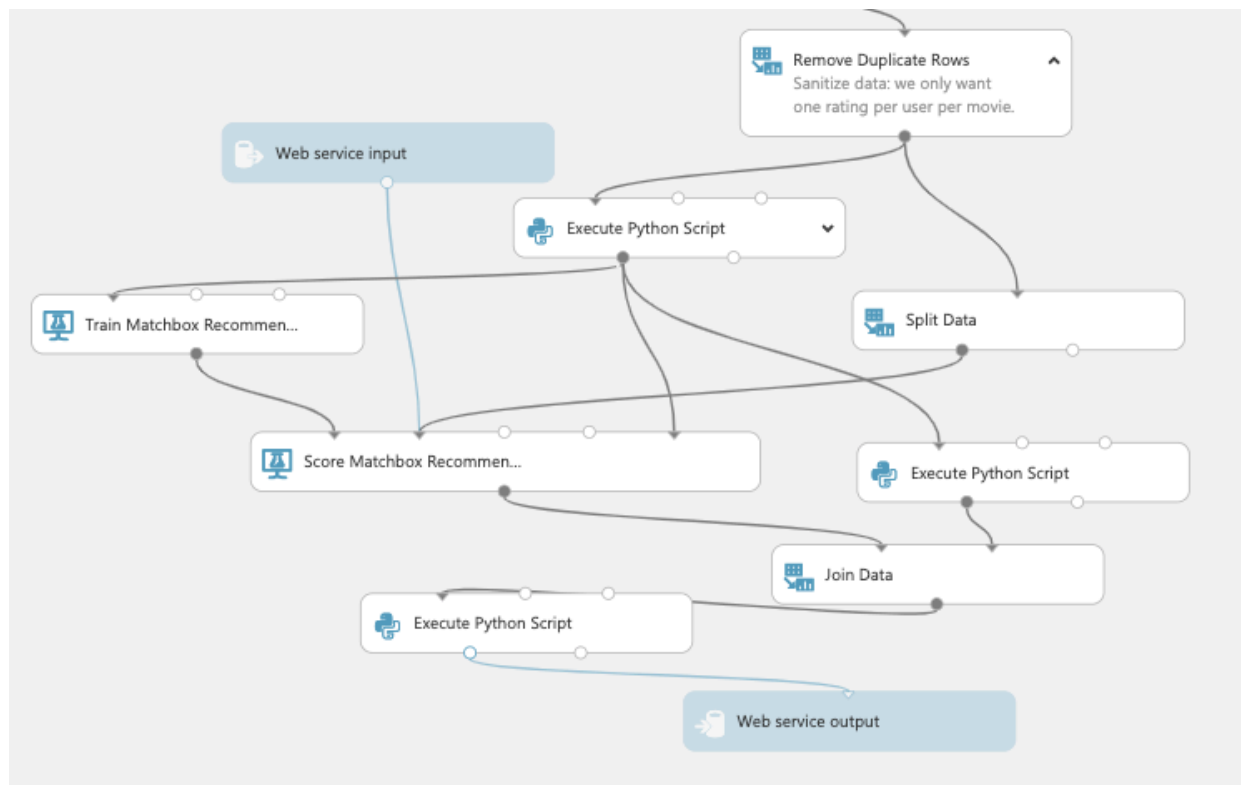
External API with Azure Machine Learning Studio

For our next API we have used the deployment services from Azure Machine Learning Studio. A GUI-based integrated development environment for constructing and operationalizing Machine Learning workflow on Azure. It is the predecessor of Azure's Machine Learning Service. We have used the Studio as it offer free deploying capabilities.

First, we created a new experiment and we uploaded our ratings and movies dataset. In the initial steps we performed some pre-processing steps:



In the second phase of the experiment we trained the matchbox algorithm; a large scale online Bayesian recommender developed by Microsoft. With several Python Scripts we manipulated the recommendations from the algorithm and we kept the those with the highest number of total ratings from previous users. The Web Service modules actually work as the parser and the responder of the API:



The experiment can be accessed from the following link:

<https://gallery.cortanaintelligence.com/Experiment/Movie-Recommender-API>

With our experiment completed, we request to deploy it on the web. This was done with the functionality “Set up Web Service” from the bottom navigation bar of the experiment. Azure generated a wide range of documentation to call our API:

prod:recommender: movie recommendation

DASHBOARD CONFIGURATION

General [New Web Services Experience](#) preview

Published experiment

[View snapshot](#) [View latest](#)

Description

No description provided for this web service.

API key

JAIPHbGN5FKDEk38mc58GJjcv8iub7/fjQCxlc+io7GEQYauL7J7AoXbHWUJsNTKBywwXYYGILud1G4uiREMKg==

Default Endpoint

API HELP PAGE	TEST	APPS
REQUEST/RESPONSE	Test Test preview	Excel 2013 or later Excel 2010 or earlier workbook
BATCH EXECUTION	Test Test preview	Excel 2013 or later workbook

In the Request/Response section we were able to extract also information on how to make a request through Python:

Sample Code

```
C# Python R Select sample code

import urllib2
# If you are using Python 3+, import urllib instead of urllib2

import json

data = {

    "Inputs": {

        "input1": {

            "ColumnNames": ["userId", "title", "rating"],
            "Values": [ [ "0", "value", "0" ], [ "0", "value", "0" ], ]
        },
        "GlobalParameters": {

    }

    }

}

body = str.encode(json.dumps(data))

url = 'https://ussouthcentral.services.azureml.net/workspaces/10666c0c4ab84aaf8c65025dcc8d9362/services/31462a707b714d2bbc6e0077a19fa13b/exe
api_key = 'abc123' # Replace this with the API key for the web service
headers = {'Content-Type': 'application/json', 'Authorization': ('Bearer ' + api_key)}

req = urllib2.Request(url, body, headers)
```

To ensure that our API works as expected, we performed a test with Postman:

The screenshot displays the Postman interface for a POST request. The URL bar shows the endpoint: `https://ussouthcentral.services.azureml.net/workspaces/10666c0c4ab84aaf8c65025dcc8d9362/se ...`. The 'Body' tab is selected, showing a JSON payload with an array of movie titles and ratings. The 'Send' button is visible. Below the request, the 'Body' tab of the response is shown, displaying a JSON object with a 'Results' field containing an 'output1' object with a 'table' type and a list of items.

```
{
  "Inputs": {
    "input1": {
      "ColumnNames": ["userId", "title", "rating"],
      "Values": [
        [ "11", "Heat (1995)", "5" ],
        [ "11", "GoldenEye (1995)", "3" ],
        [ "11", "Dead Man Walking (1995)", "4" ],
        [ "11", "Mortal Kombat (1995)", "2" ],
        [ "11", "Broken Arrow (1996)", "3" ],
        [ "11", "Braveheart (1995)", "5" ],
        [ "11", "Apollo 13 (1995)", "5" ],
        [ "11", "Batman Forever (1995)", "3" ]
      ]
    }
  }
}
```

```
{
  "Results": {
    "output1": {
      "type": "table",
      "value": {
        "ColumnNames": [
          "User",
          "Item 1",
          "Item 2",
          "Item 3",
          "Item 4",
          "Item 5"
        ]
      }
    }
  }
}
```

The API was used in the client's side scenario.

Client's call

In the second folder of our GitHub repo we developed a script file which calls the several APIs that we have created. In the beginning we load the users that have been assigned for test use and then we collect their recommendations from the three APIs that we have made. For these user-movie pairs we request scores from the SageMaker API that was provided from the teaching team for this project. In the exported .csv file we kept the five movies for each user with the highest score from the SageMaker API. The accompanied Jupyter Notebook provides a breakdown of the calls in the different APIs.

Suggestions for further Improvement

Throughout the project we have discovered several areas that could be further improved. First of all, we have not focused on this project in the accuracy of our APIs, nor in the ensemble of their recommendations. For some test-users we could have used only some of the movies that they have rated and keep the rest for validation purposes. In addition, with the exception of the first API we haven't focus on the execution time of the APIs or in the training/predicting time of the models. Moreover, we could have devised strategies to make the several queries to the different APIs in parallel and not in a sequence. We haven't explored other type of recommender system types (for example content-based approaches). Finally, due to time limitations, we haven't managed to provide recommendations with a format of a .csv file as it was requested.

References

1. Dianas JA. jadianes/spark-movie-lens [Internet]. 2020 [cited 2020 Apr 28]. Available from: <https://github.com/jadianes/spark-movie-lens>
2. Kim R. Deploying PySpark ML Model on Google Compute Engine as a REST API [Internet]. Medium. 2018 [cited 2020 Apr 29]. Available from: <https://towardsdatascience.com/deploying-pyspark-ml-model-on-google-compute-engine-as-a-rest-api-d69e126b30b1>
3. Liao K. Prototyping a Recommender System Step by Step Part 2: Alternating Least Square (ALS) Matrix... [Internet]. Medium. 2018 [cited 2020 Apr 29]. Available from: <https://towardsdatascience.com/prototyping-a-recommender-system-step-by-step-part-2-alternating-least-square-als-matrix-4a76c58714a1>
4. Sapphirine/Recommendation-on-Google-Cloud-Platform [Internet]. Sapphirine Big Data Repositories; 2017 [cited 2020 Apr 28]. Available from: <https://github.com/Sapphirine/Recommendation-on-Google-Cloud-Platform>
5. FAQ — Surprise 1 documentation; How to get the top-N recommendations for each user [Internet]. [cited 2020 Apr 28]. Available from: <https://surprise.readthedocs.io/en/stable/FAQ.html?highlight=get%20top%20predictions#how-to-get-the-top-n-recommendations-for-each-user>
6. Matchbox: Large Scale Bayesian Recommendations - Microsoft Research [Internet]. [cited 2020 Apr 29]. Available from: <https://www.microsoft.com/en-us/research/publication/matchbox-large-scale-bayesian-recommendations/?from=https%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F79460%2Fwww09.pdf>
7. MLflow Documentation — MLflow 1.8.0 documentation [Internet]. [cited 2020 Apr 28]. Available from: <https://mlflow.org/docs/latest/index.html>
8. MovieLens | GroupLens [Internet]. [cited 2020 Apr 28]. Available from: <https://grouplens.org/datasets/movielens/>
9. The Postman Platform [Internet]. [cited 2020 Apr 28]. Available from: <https://www.postman.com/postman>
10. Welcome to Flask — Flask Documentation (1.1.x) [Internet]. [cited 2020 Apr 28]. Available from: <https://flask.palletsprojects.com/en/1.1.x/#>