MSIN0166: Data Engineering

Individual Coursework

Symeon Kokovidis
18116815

29th April 2020

Table of contents

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 combability 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 surpise 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 × 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

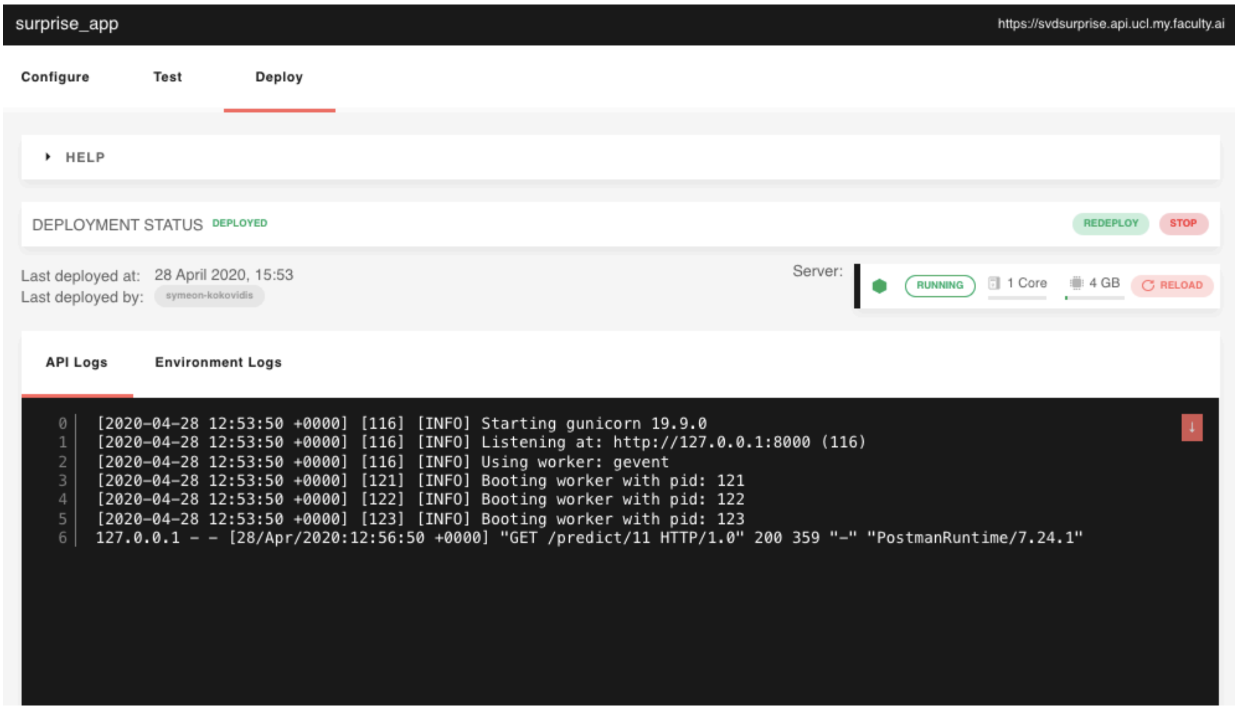
Servers' setup

Internal API in Faculty.ai

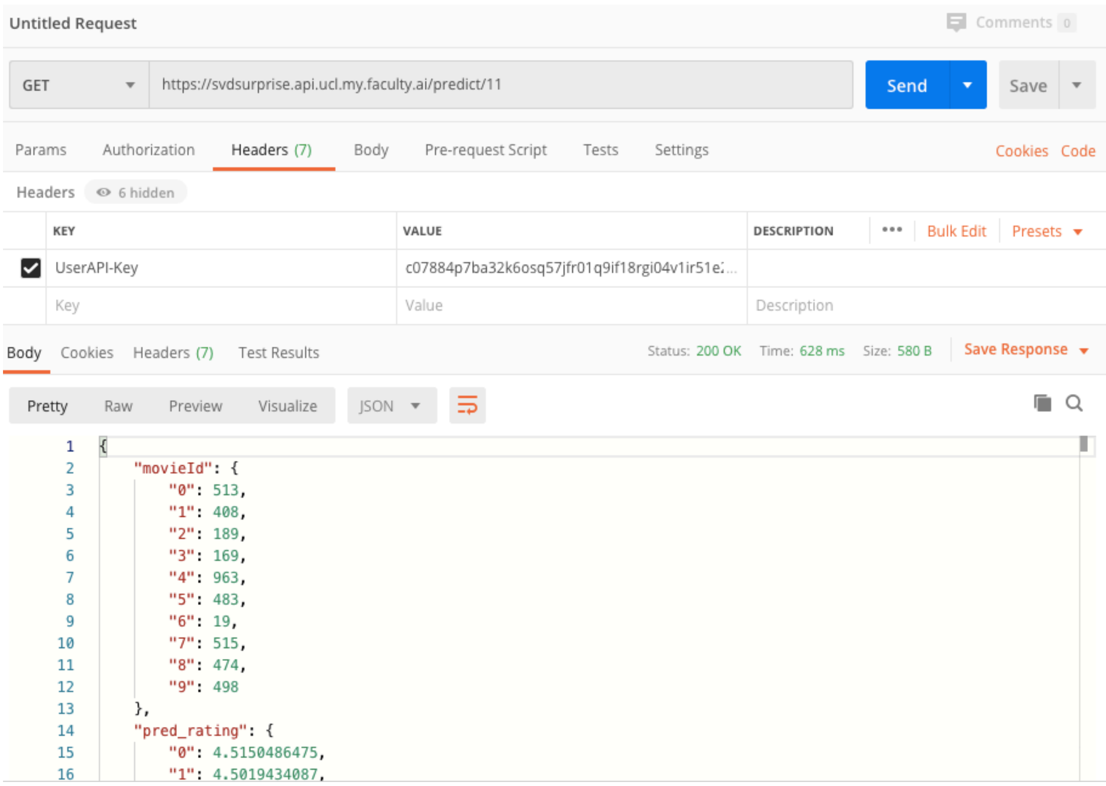
For the selection of the model for our first API, we have used the surprise recommender systems library with MLflow. We examined 10 different recommender algorithms and we logged their performance with MLflow. MLflow actually stored the training/predicting time and their accuracy (Root Mean Square Error). 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.



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



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.

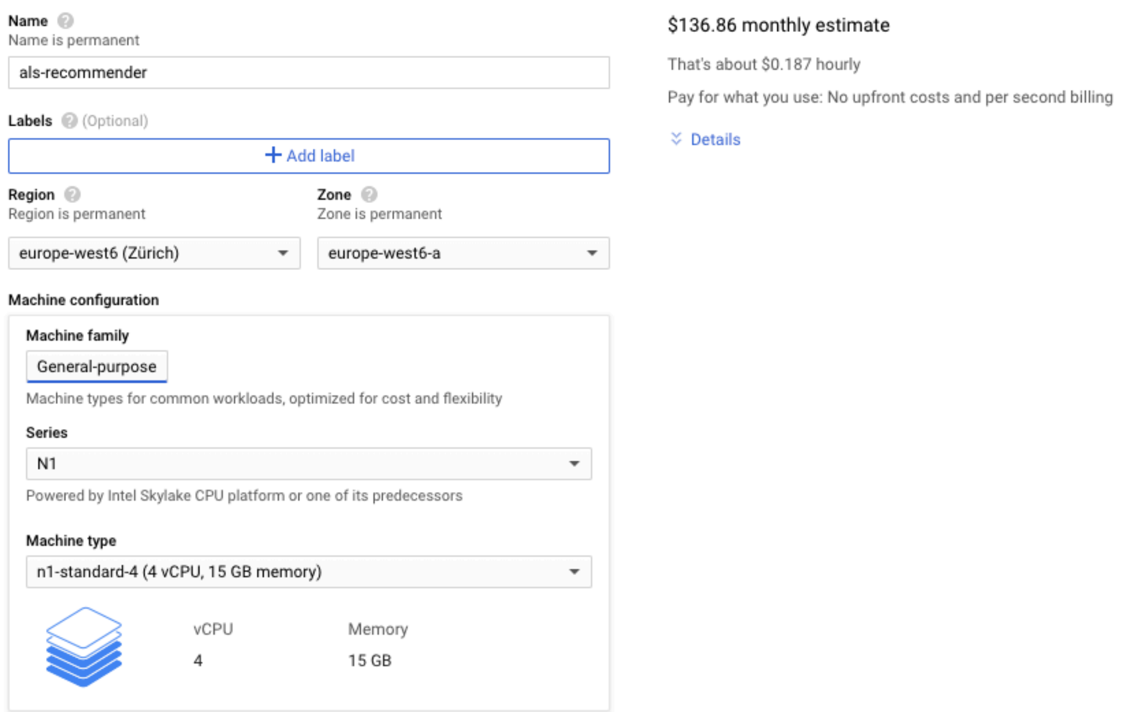
- The initialize_data.py actually removes the test users (and their ratings) that we are going to get recommendations for them in the Client's setup chapter.

- 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

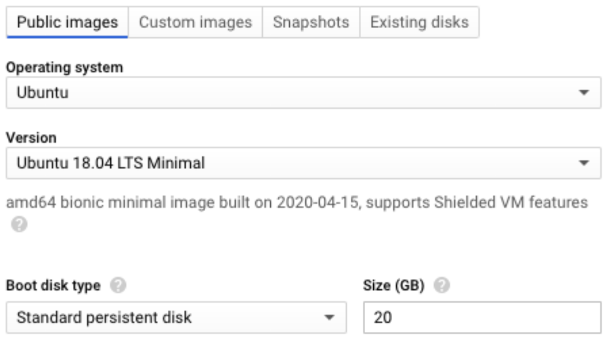
- The app.py actually holds our Flask server and our POST API. In more detail, the script loads the existing ratings , parses a new user's ratings, retrains the ALS algorithm (with the best hyperparameters from cross_validation.py) and returns movie recommendations for the user with a .json file. These 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.

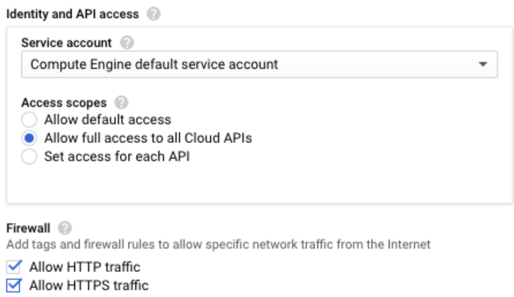
The initial deployment of the app was done in a Virtual Machine on Google Cloud Platform. We have selected a region which will be close to our potential clients



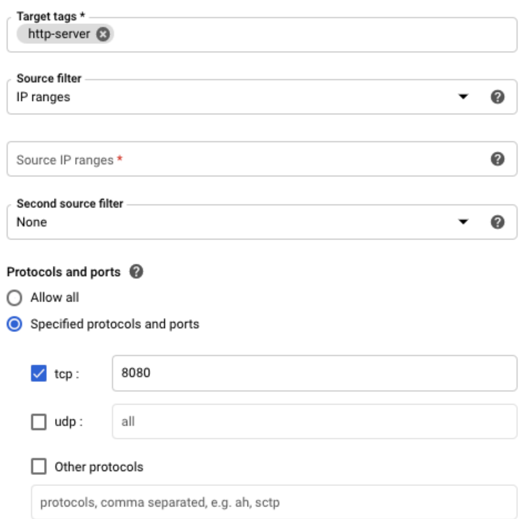
For the Operating System we used the Ubuntu 18.04 LTS Minimal version:



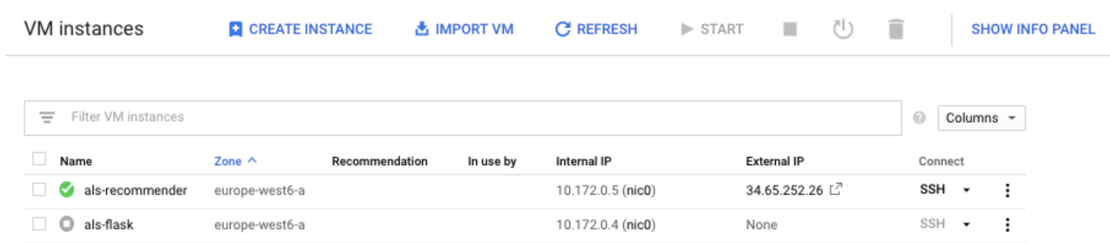
We requested full access to all cloud APIs and we allows HTTP/HTTPS traffic:



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 :



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



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'
```

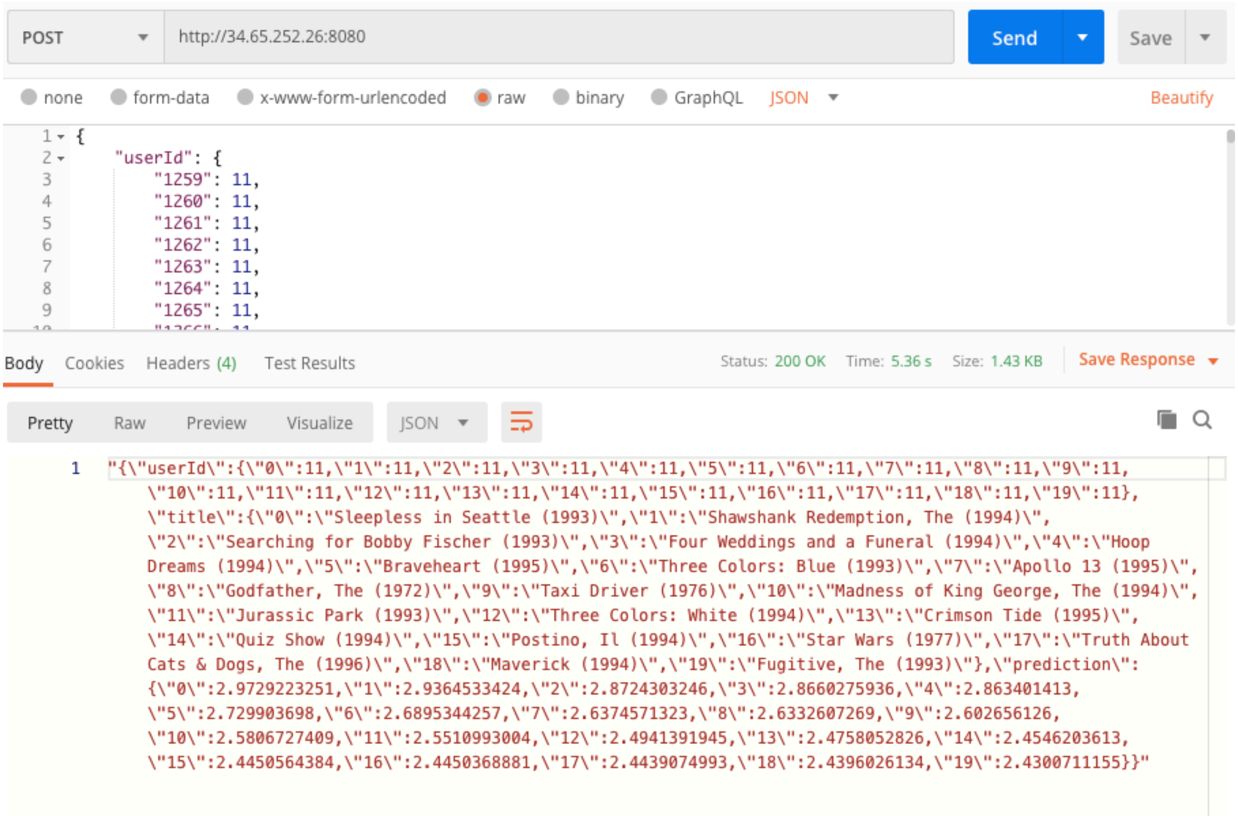
5. Executed the three script files

```
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
```

At this stage a client can access the API. With Postman we perform a test query:



Note that the API gets the server's IP address

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

First we install docker. The docker_install shell scripts contains all the required commands

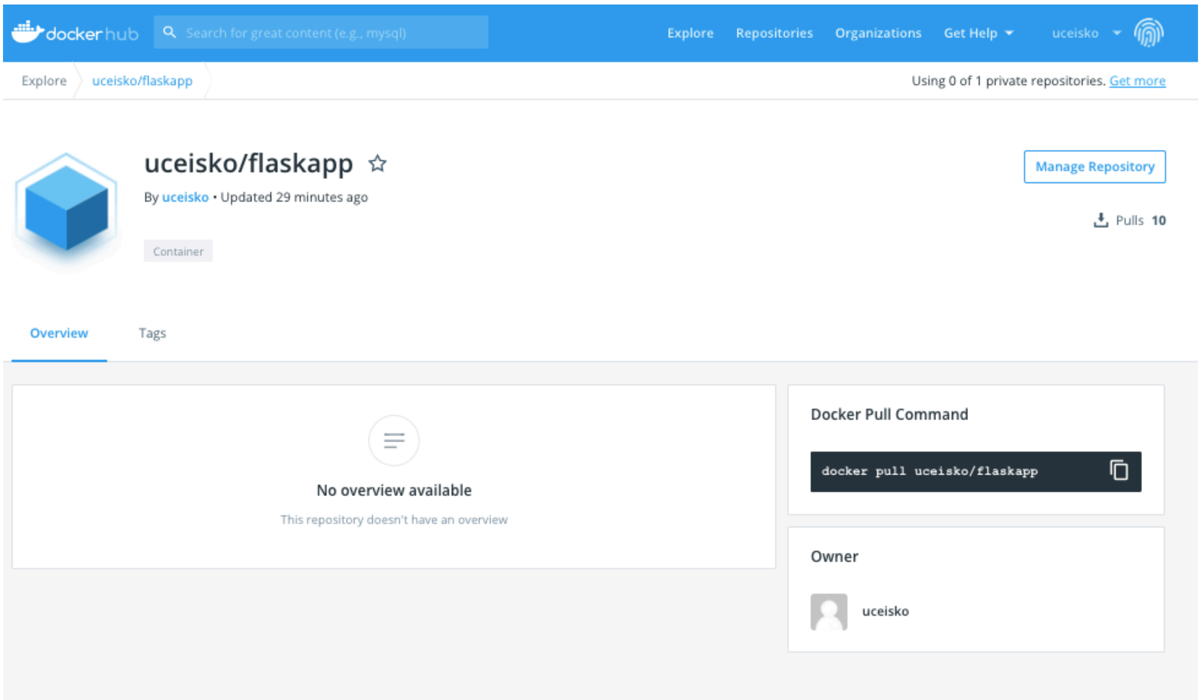
```
sudo bash ./docker_install.sh
```

Then we have created an image with our working directory. The build_repo_docker_hub scripts uses the Dockerfile (located in the same directory) to create the image. The same script files logins to docker where we will need to pass our credentials. After the successful login, push the image to a Docker Hub repository:

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

When the push starts we should see the following process:

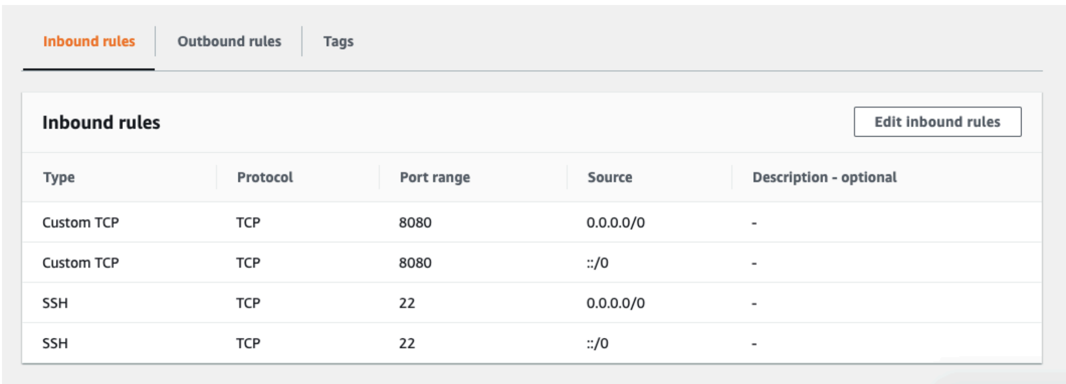
After the successful commit we should be able see our repo on Docker Hub:



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

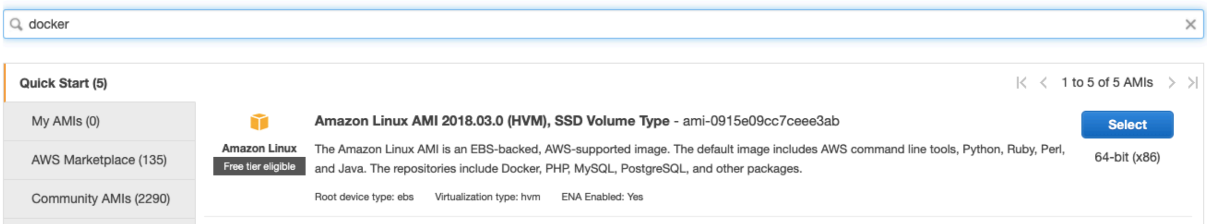
In the next steps we show how we pulled the repo to a new Virtual Machine on Amazon Web Services.

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

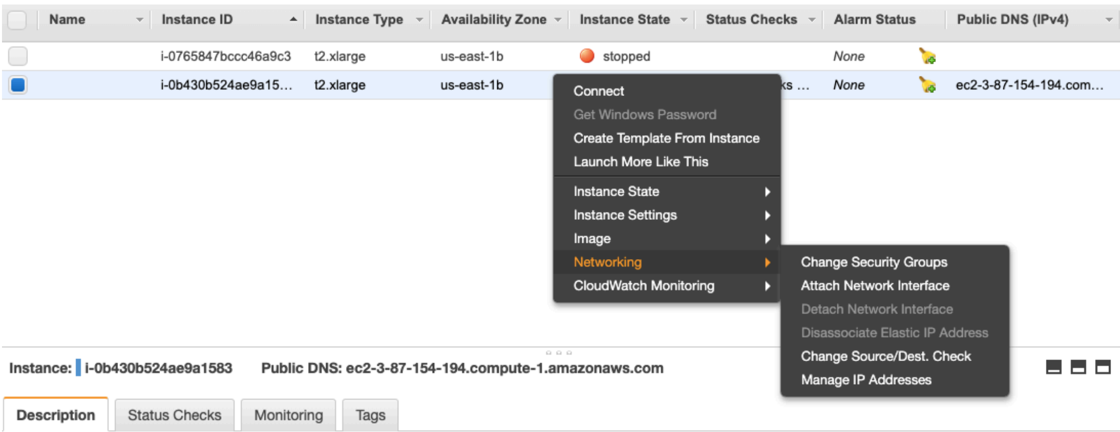


Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	8080	0.0.0.0/0	-
Custom TCP	TCP	8080	::/0	-
SSH	TCP	22	0.0.0.0/0	-
SSH	TCP	22	::/0	-

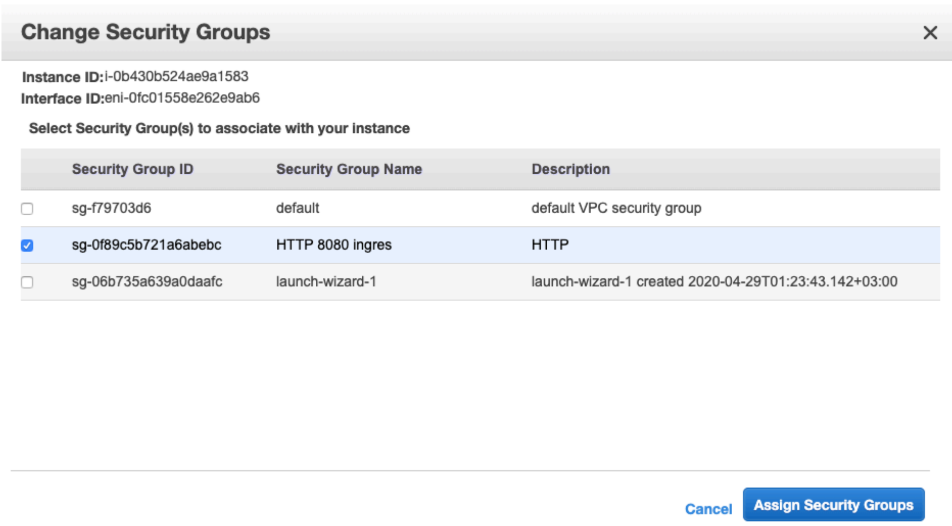
Then, we launched a new instance with docker pre-installed:



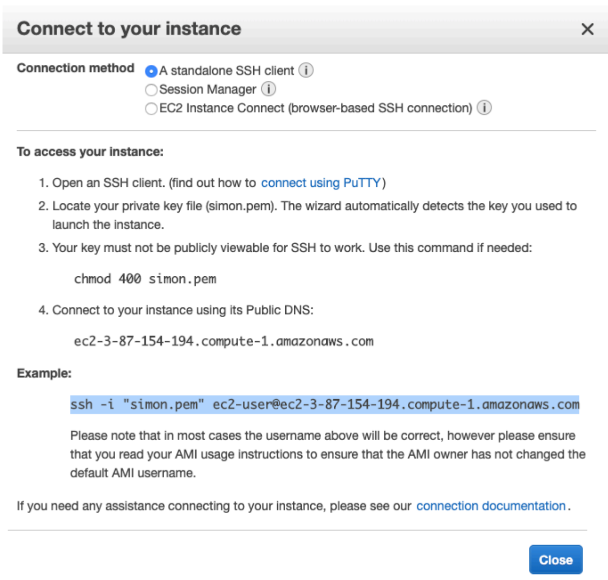
Before we access the instance we changed its security group to the one that we have created. We right clicked on the instance, and we access the Security Groups:



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



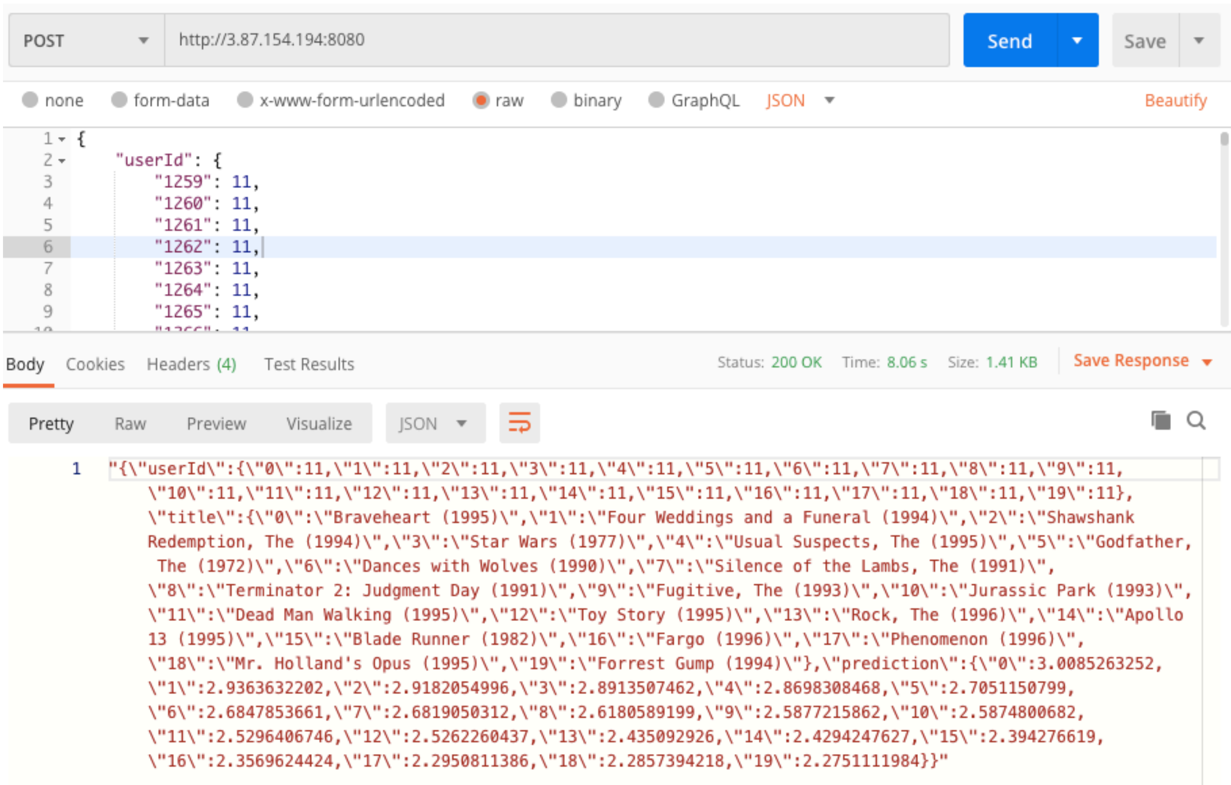
With Mac OSX terminal we accessed the instance:



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:

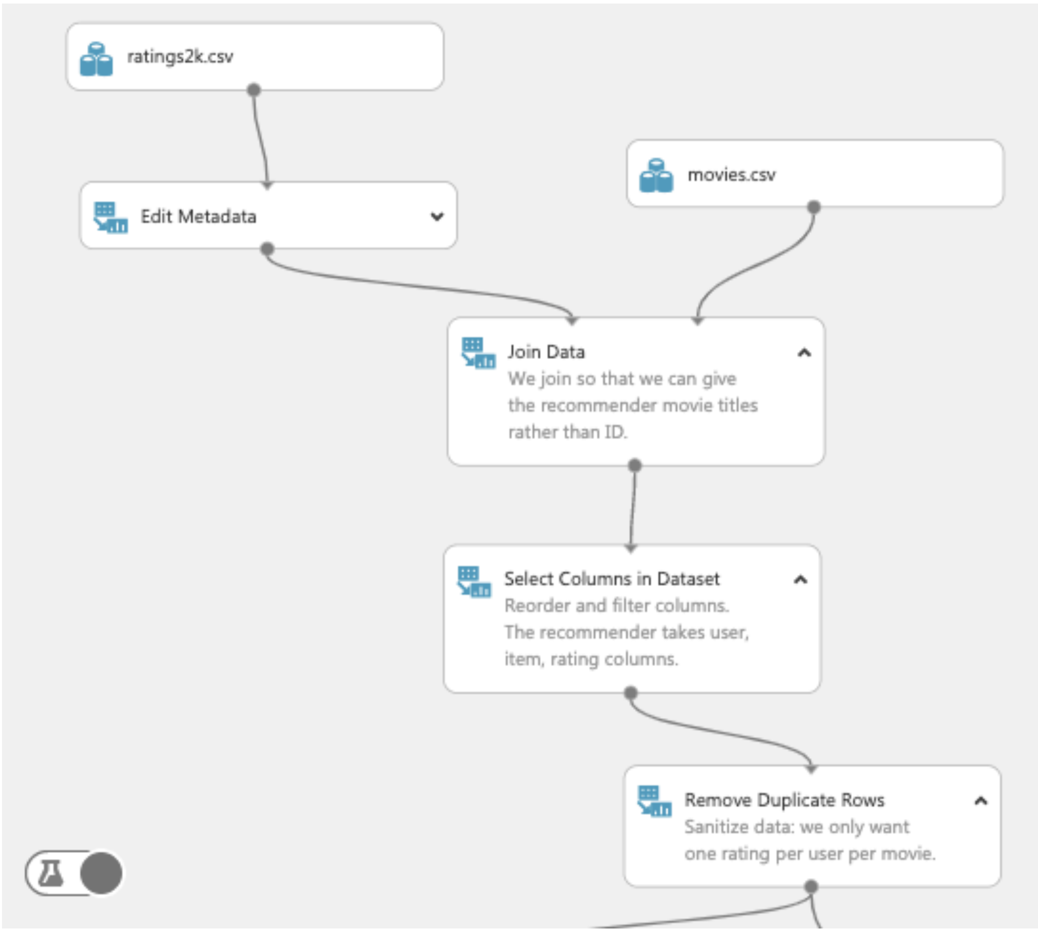


Note that in this request, we have changed the IP address to the one of our new instance.
This API will be used in our client's side scenario.

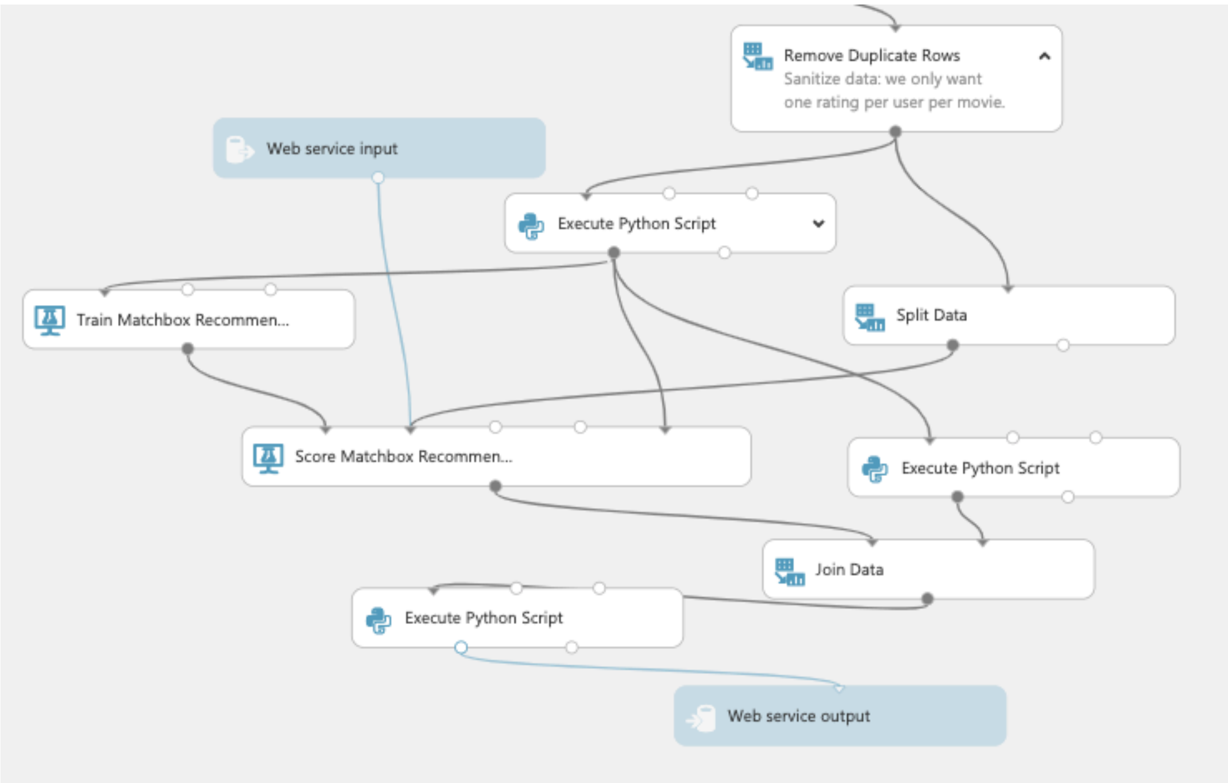
External API with Azure Machine Learning Studio

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:

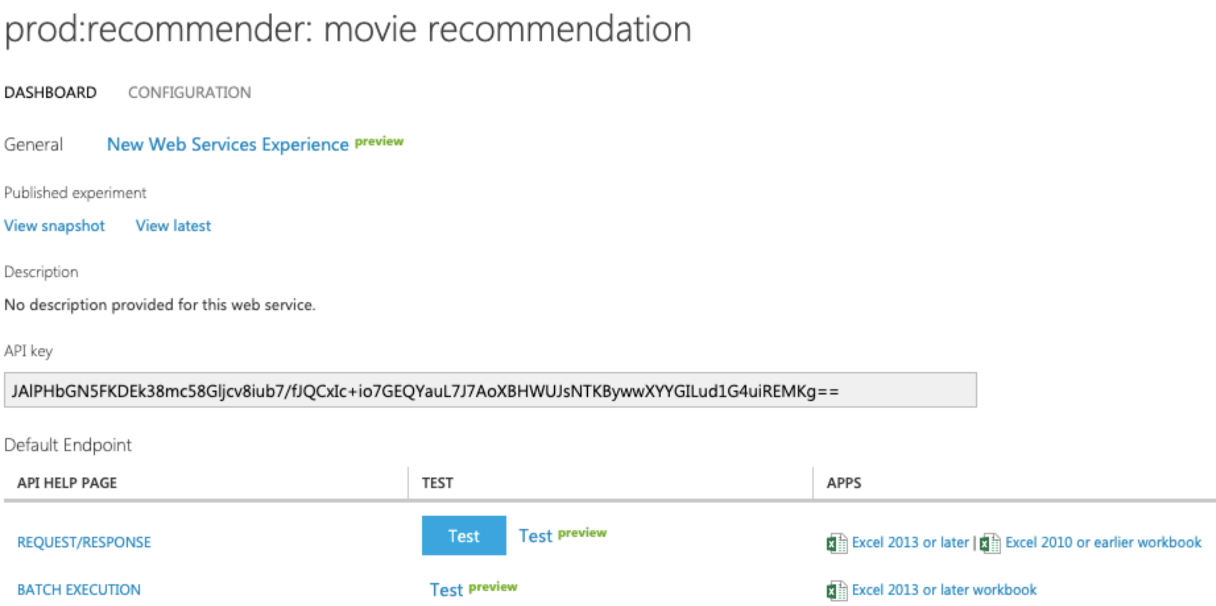


In the second phase of the experiment we trained the matchbox algorithm; a large scale online Bayesian recommender developed by Microsoft. With several Python Scripts we manipulated the recommendations from the algorithm and we kept the those with the highest number of total ratings from previous users. 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:



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

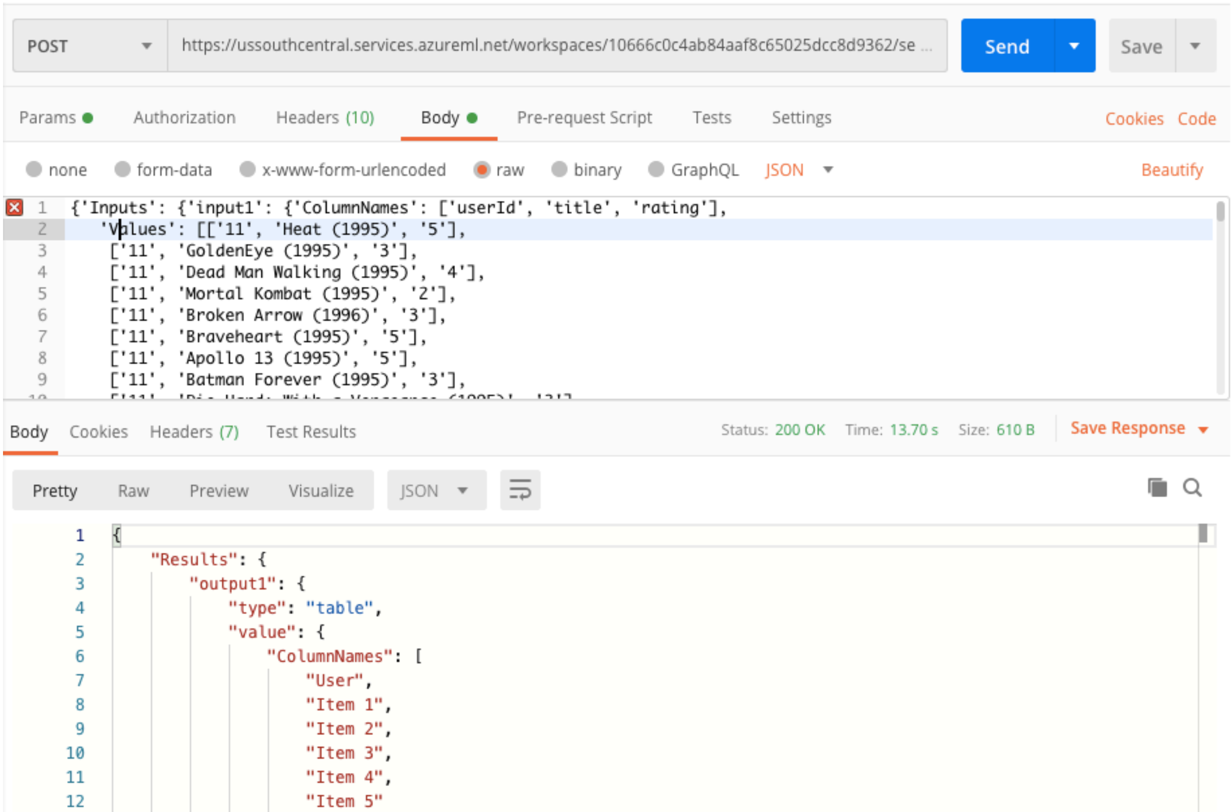
Sample Code

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 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. Dianes 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/#