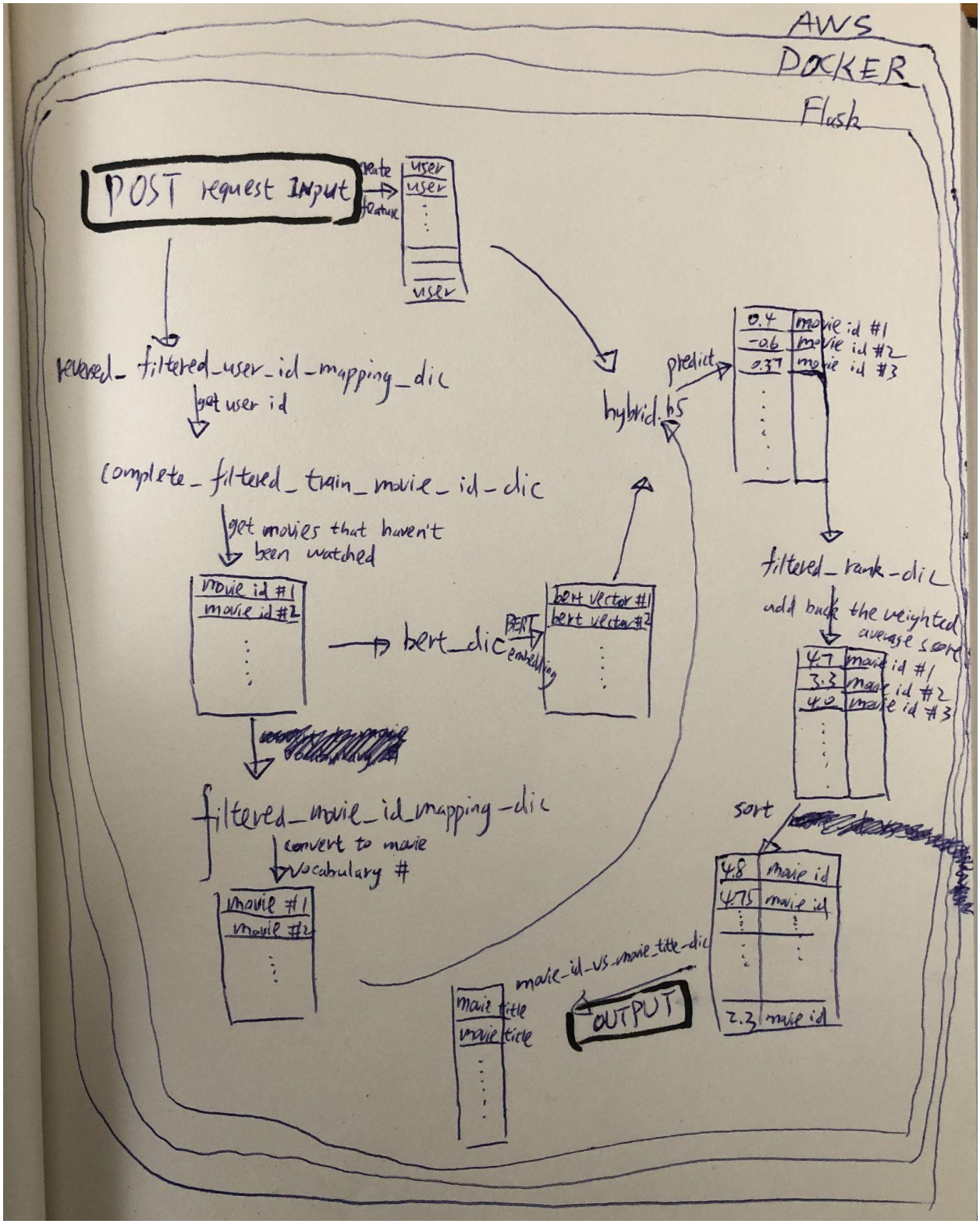
AWS
DOCKER
Flask
POST request Input
create
feature
user
user
user
reversed_filtered_user_id_mapping_dic
get user id
complete_filtered_train_movie_id_dic
get movies that haven't been watched
movie id #1
movie id #2
bert_dic
BERT embedding
bert vector #1
bert vector #2
filtered_movie_id_mapping_dic
convert to movie vocabulary #
movie #1
movie #2
hybrid.h5
predict
0.4
-0.6
0.37
movie id #1
movie id #2
movie id #3
filtered_rank_dic
add back the weighted average score
4.7
3.3
4.0
movie id #1
movie id #2
movie id #3
sort
4.8
4.75
movie id
movie id
2.3
movie id
movie_id_vs_movie_title_dic
movie title
movie title
OUTPUT

==========

A. Why I chose to design my system in the way depicted by the diagram:

The system is meant to mainly make movie recommendations for users that are already in the training set. Therefore, there is no need to re-train the model. This makes the deployment relatively straight forward. For someone who intends to use our movie recommender API, this person just needs to send a curl post request on the terminal following this article: "https://towardsdatascience.com/simple-way-to-deploy-machine-learning-models-to-cloud-fd58b771fdcf".

The curl post request should only send a string of integer representing the vocabulary number (related to user embeddings) for the user whom we're trying to make recommendations for. The vocabulary number is then used as a key for the python dictionary "reversed_filtered_user_id_mapping_dic" to find the corresponding user id (a string). The user id is then used as a key for the python dictionary "complete_filtered_train_movie_id_dic" to find all the movie ids that haven't been watched by this user. These movie ids are then mapped to their corresponding movie vocabulary numbers using the python dictionary "filtered_movie_id_mapping_dic". On the other hand, these movie ids are also mapped to their corresponding movie overview embedding vectors using the python dictionary "bert_dic". Each embedding vector has 768 components. We then make m copies of the user vocabulary number where m is the number of movies that haven't been watched by this user. Feeding the user vocabulary feature (a numpy array of shape (m,)), the movie vocabulary feature (a numpy array of shape (m,)) and the movie bert vector feature (a numpy array of shape (m, 768)) to the Keras model (loaded from the file "hybrid.h") we get the predictions. The predictions are not yet the predicted movie ratings because we need to add back the weighted average rating scores of the corresponding movies using the python dictionary "filtered_rank_dic". We then sort the movies by their predicted scores from high to low and we get the movie ids for the top 10 movies with the highest predicted ratings. Eventually we map these movie ids to the corresponding movie names using the python dictionary "movie_id_vs_movie_title_dic".

In the case where we want to make movie recommendations for a new user, we will simply output the top 10 movies with the highest weighted average movie rating recorded in the python dictionary "filtered_rank_dic". **This case is not depicted in the hand-drawn diagram.**

A. Why I have chosen to use certain technologies.

Following the amazing article quoted at the beginning of this document, we first “wrap the inference logic into a flask web service”, we then “use Docker to containerize the flask service”, finally we “host the docker container on an Amazon ec2 instance”. The reason why we chose this approach is that this is a typical and straightforward way to deploy our movie recommender deep learning model to production “with a simple tech stack”.

B. Include an estimate of the cost of the system (e.g. computing resources, time, money).

In terms of the cost and budget, we will use “tx.xlarge” (16 GB) which has an on-demand cost of 0.1856 USD/hour.

==========

Pre-deployment checklist

1. Define the problem

Given user-movie ratings and movie overviews (paragraph of texts), build a movie recommender system that predicts movie ratings for unrated movies by all users in the training set. Making good recommendations is important for companies like Netflix to grow and keep their customers. To manually make recommendations one could recommend movies watched by users of similar tastes.

2. Prepare the data

Available data are user-movie ratings, movie overviews as well as some other metadata of movies. There might be other things that are available. It could be useful to have the actual date when the movies are watched or/and rated.

We could train on a subset of the training data by e.g. only using a certain percentage of the total training data and filter out rarely rated movies and users who don’t give enough ratings. The most useful attribute is probably the user-movie ratings.

We need to map the user and movie ids to vocabulary numbers for user and movie embeddings. We also need to get movie overview embeddings by feeding the movie overview texts to BERT.

3. Spot check algorithms

The cross validation and test sets are randomly selected from the user-movie ratings. They are only a small percentage of the whole data. This way it will be likely that all movies and users in the cross validation and test sets are included in the training set. We evaluated the model on metrics like RMSE, MAE and mean average precision.

4. Algorithm tuning

We could tune learning rate, batch size, model architecture, layer size, regularization parameters, etc. We also used a hybrid model that combines collaborative filtering method and content based method.

5. Present the project

The model evaluation is presented at the end of the corresponding notebook(s) on my github :https://github.com/fanglidayan/machine_learning_projects/blob/master/capstone_project/10-netflix-movie-recommender-part-3b.ipynb.

The model deployment files can be found in the following github repo: https://github.com/fanglidayan/machine_learning_projects/tree/master/capstone_deployment_git.