### **CHAPTER 9: Working with Amazon SageMaker**

### **- 41 pages**

### Data scientists spend 80% of their time wrangling, cleaning, and featurizing data before a model is fit to the prepared dataset. If parts of this data preparatory process can be automated, it provides a huge lift in terms of the number of problems that data scientists and companies can solve. SageMaker from Amazon Web Services is the product/service that exactly solves this problem.

### It is a fully managed service that enables data scientists to build, train and deploy machine learning models seamlessly. SageMaker also provides an integrated Jupyter environment where data scientists can conduct data exploration. It brings productivity and efficiency to data science lifecycle by managing infrastructure provisioning, scaling and tear down. The platform offers tools, such as experimentation service and hyperparameter tuning optimization, to manage experiments and reduce the time required to arrive at an optimal model configuration.

The design pattern to execute models in SageMaker is to read the data placed in S3. The data may not be readily consumable most of the items. If the datasets required are large, then wrangling the data in the Jupyter notebook may not be practical. In such cases, Spark EMR clusters can be employed to conduct operations on big data. The resulting datasets can be placed in S3 for SageMaker to consume. After the data is transformed into a format required by the algorithm, training jobs can be created to fit model to the data. The trained model can then be used to deploy an endpoint, which can be used to run inferences on unseen data. As you can see, data preparation, model training and deployment stages of the lifecycle can all be automated with SageMaker, without having to worry about the underlying infrastructure complexities.

What will we cover in this chapter:

* Process large scale data through Spark EMR cluster
* Conduct training in Amazon SageMaker
* Run hyperparameter optimization
* Understand SageMaker Experimentation Service
* Model deployment and inference
* Bring Your Own Model – SageMaker, MXNet, Gluon
* Bring Your Own Container – R model

For the next three sections, we will focus on book ratings dataset. The dataset consists of ratings from over 100k book lovers. More details on this dataset can be found at  [Book-Crossing Dataset.](http://www2.informatik.uni-freiburg.de/~cziegler/BX/" \t "_blank)

#### We are going to explain each of the capabilities of Amazon SageMaker through few tailored datasets. In the model training section of the book, we will illustrate how recommender system can be built with ease in Amazon SageMaker

#### **Big data preprocessing using Spark EMR**

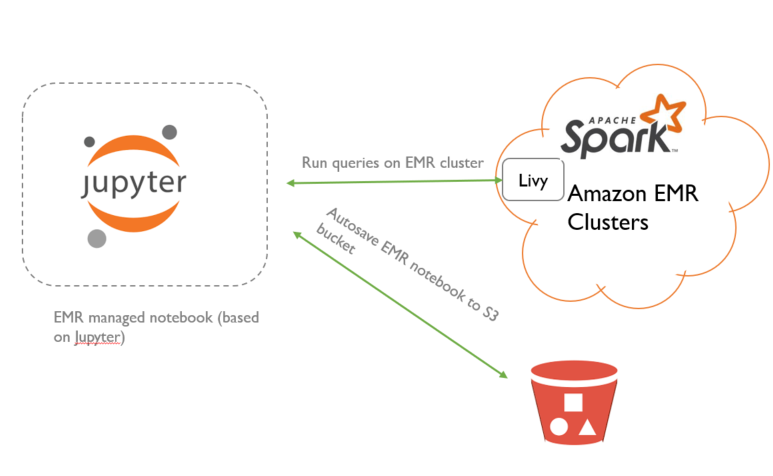
In the folder associated with this chapter, you will find 2 csv files:

* BX-Book-Ratings.csv – contains book ratings; User ID; ISBN; Rating
* BX-Books.csv – contains book attributes, including title

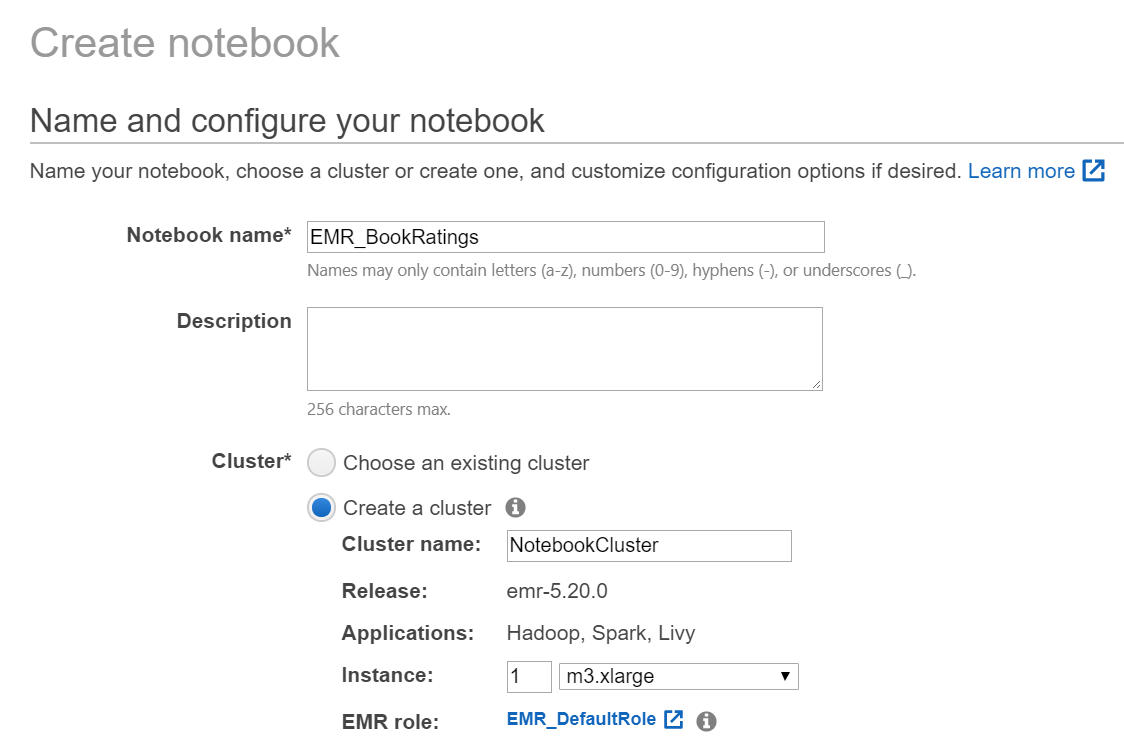
The dataset contains a total of 1.5 million ratings and 340k books.

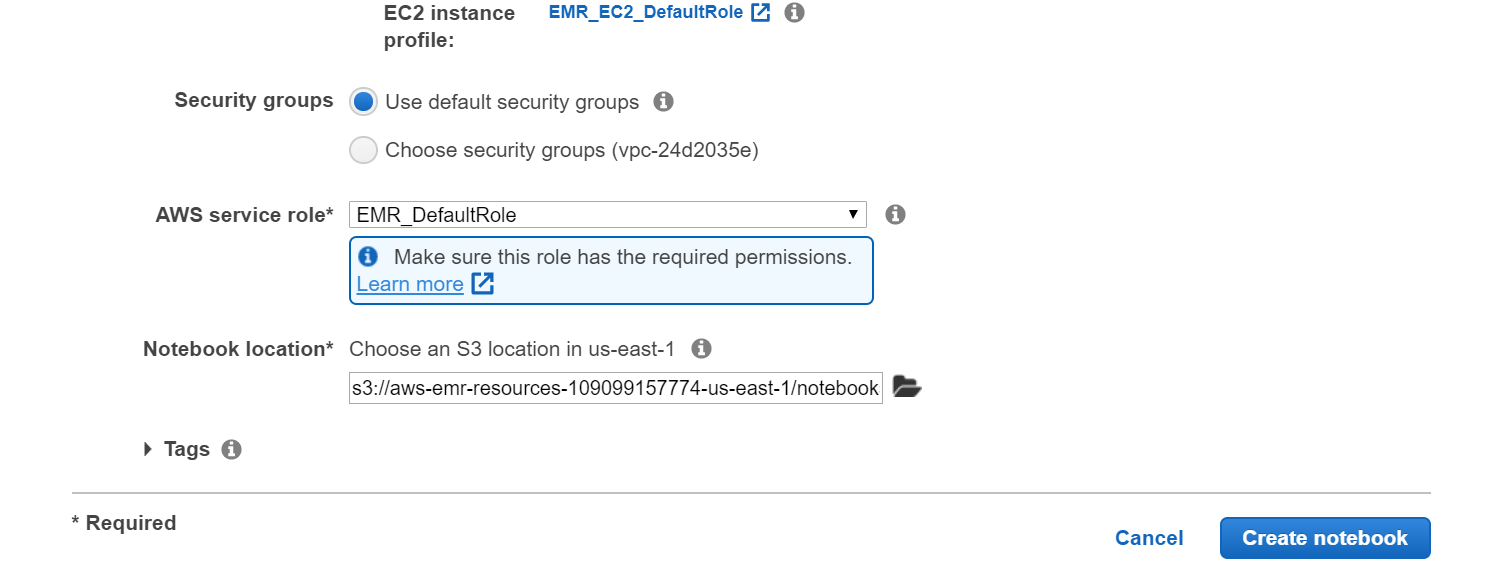
Wrangling this big dataset in Jupyter notebooks results in out-of-memory errors. Our solution is to employ AWS EMR (Elastic Map Reduce) clusters to conduct distributed data processing. Hadoop will be used as the underlying distributed file system, while Spark will be used as the distributed computing framework. Livy service, which is an open source REST interface for interacting with Spark clusters without the need for Spark client, will also be installed as an application on the EMR cluster. The Livy service enables the communication between EMR notebook and the EMR cluster.

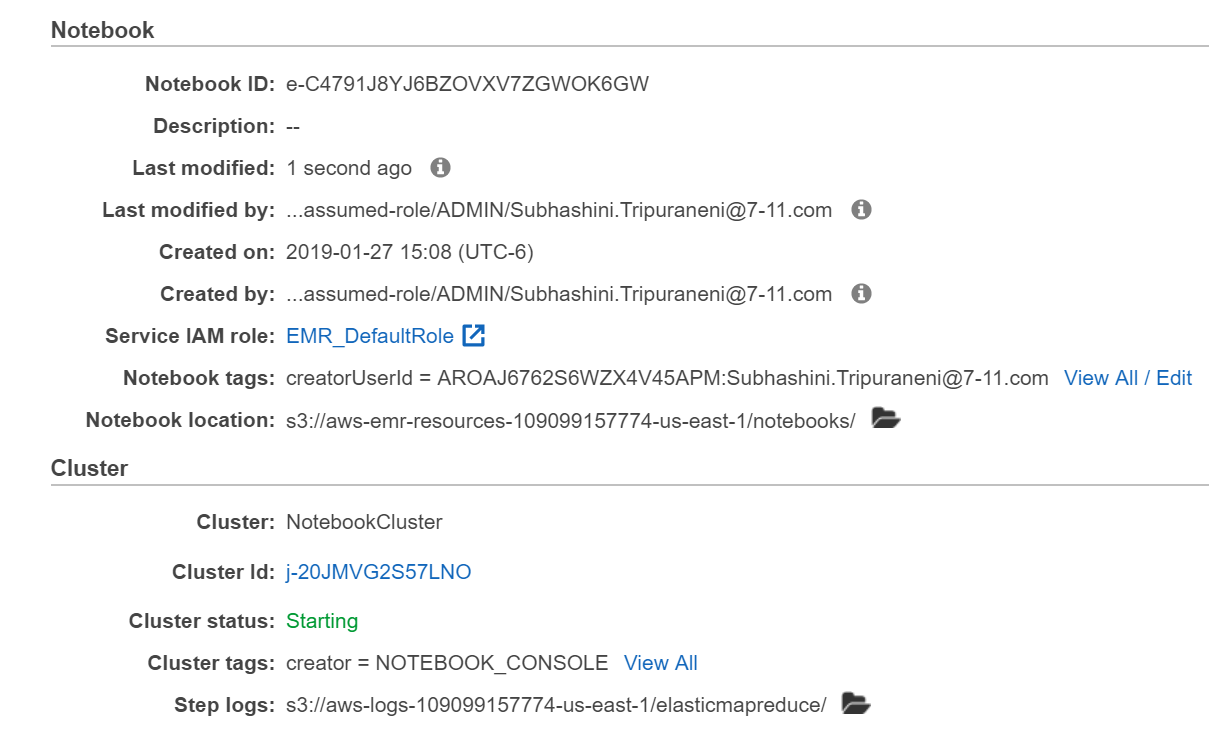
Now, to run commands against the EMR cluster to process big data, AWS offers EMR notebooks. EMR notebooks provide a managed notebook environment, based on Jupyter notebook. These notebooks can be used to interactively wrangle large data, visualize the same, and prepare analytics ready datasets. Data engineers and data scientists can employ a variety of languages, Python, SQL, R and Scala, to process large volumes of data. These EMR notebooks can also be saved periodically to a persistent data store, S3, so the saved work can be retrieved later.

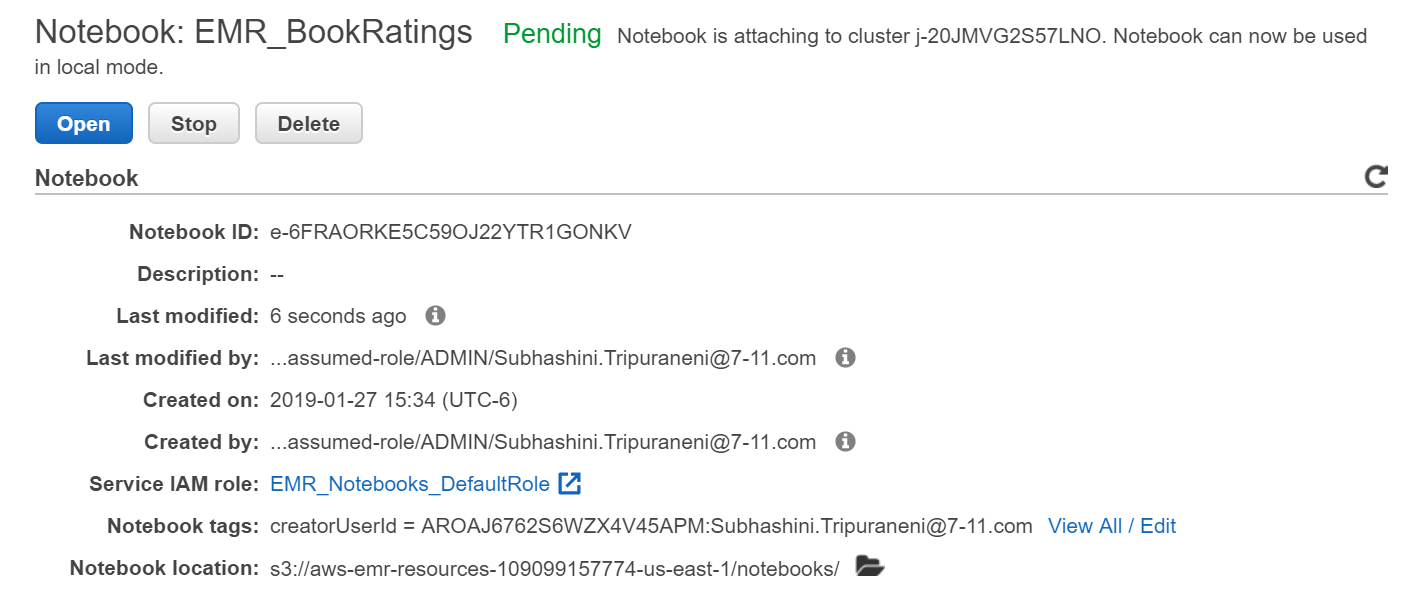
Below is the architecture diagram, detailing how EMR notebooks communicate with Spark EMR clusters to process large data 

Let’s begin by spinning up Spark EMR cluster, configure appropriate security groups to allow communication between EMR notebook and master node of EMR cluster. We will also assign a service role to EMR cluster, so it can interact with other AWS services.









The cluster has to be created first before the notebook can be attached to it. Once the cluster goes into ‘Waiting’ status, the notebook can be associated with it.

After the status of the notebook changes to ‘Ready’, you can navigate to it

We will first obtain information about the session – we’ve launched 1 node (m3.xlarge) EMR cluster. The current session has 2 cores and pyspark kernel is running.

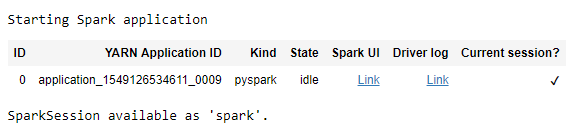
%%info

Current session configs: {'driverMemory': '1000M', 'executorCores': 2, 'kind': 'pyspark'}

Let’s define S3 bucket and the directory path within the bucket where the processed data will be stored.

s3\_bucket = 's3://ai-in-aws/'

output\_prefix = 'object2vec/bookratings'



**Read input data (csv) from S3**

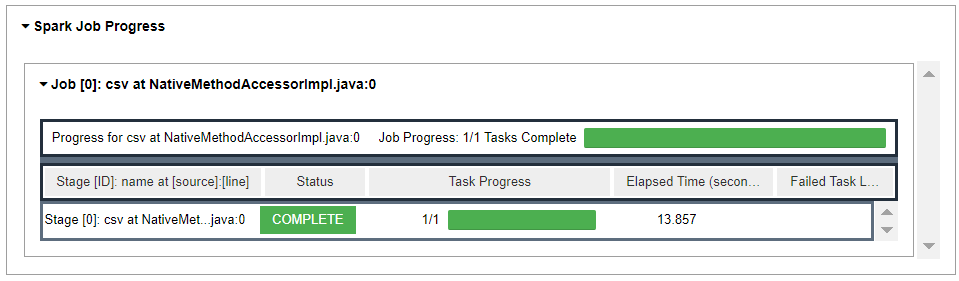
from pyspark.sql.window import Window

from pyspark.sql import functions as F

The csv datasets stored on your S3 bucket (s3://<your-bucket-name>/<folder>) can be retrieved using the *spark* session

ratings = spark.read.option("header","true").option("quote", "\"").option("delimiter", ";").csv("s3://ai-in-aws/awsglue-datasets/BX-Book-Ratings.csv")

By expanding the *Spark Job Progress*, you can review the progress of the spark job – number of tasks launched and their status



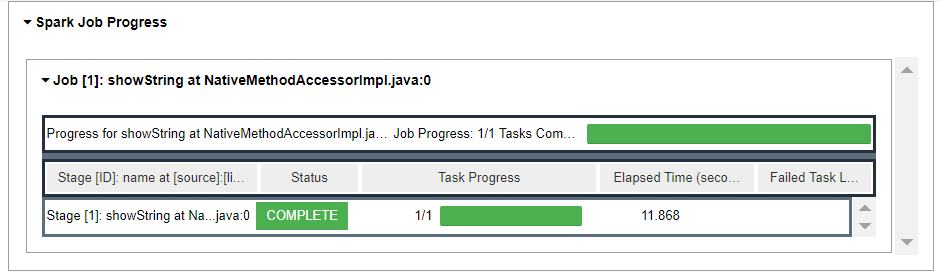
After we read the ratings dataset, let’s clean-up ISBN column, removing whitespaces and ensuring that the alphanumeric book identifier is in lowercase.

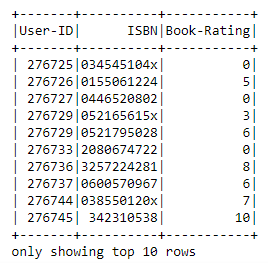
#Ensure ISBN is cleaned

ratings = ratings.withColumn("ISBN", F.lower(F.trim(F.col("ISBN"))))

Let’s look to see if the data has been correctly read.

ratings.show(10)





**Address the long tail problem in the dataset**

Books that are not rated by at least a few users and users who have not rated at least a few books cannot be modeled in terms of creating a recommender system. This is because, in this case, we will not have enough data on users and/or books

Let’s group the ratings by User and Book to determine number of books rated by a user and number of users who rated a book respectively.

def value\_counts(df, colName):

return (df.groupby(colName).count()

.orderBy('count', ascending=False))

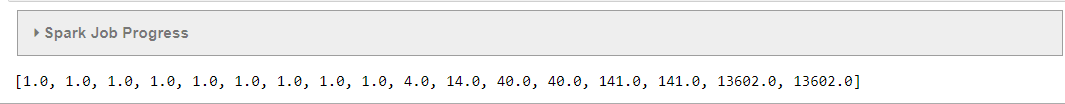
Now, let’s review the distribution of the counts by quantile.

# Number of ratings per user

# Let's pick users who have rated at least 13 books

users = value\_counts(ratings, 'User-ID')

users.approxQuantile('count', [0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.96, 0.97, 0.98, 0.99, 1.0], 0.01)



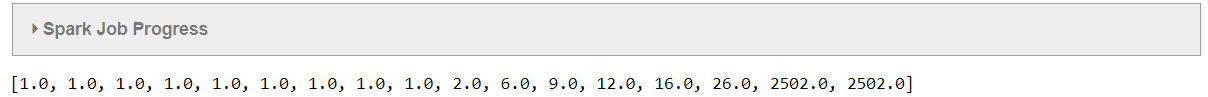
It is clear that 75% of the users rated less than or equal to 4 books.

# Number of ratings per book

# Let's pick books that have been rated by atleast 6 users

books = value\_counts(ratings, 'ISBN')

books.approxQuantile('count', [0.0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.1, 0.25, 0.5, 0.75, 0.9, 0.95, 0.96, 0.97, 0.98, 0.99, 1.0], 0.01)



Again, 75% of the books have been rated by 6 or less users

We will remove these entries from the ratings dataset to ensure we have rich data to model book our recommender system

# Filter ratings by selecting books that have been rated by at least 6 users and users who have rated at least 13 books

fil\_users = users.filter(F.col("count") >= 13)

fil\_books = books.filter(F.col("count") >= 6)

#Number of books meeting the threshold; 34780

fil\_books.count()

34780

#Number of users meeting the threshold

fil\_users.count()

10468

**Obtain book title information**

We will read book details from second dataset.

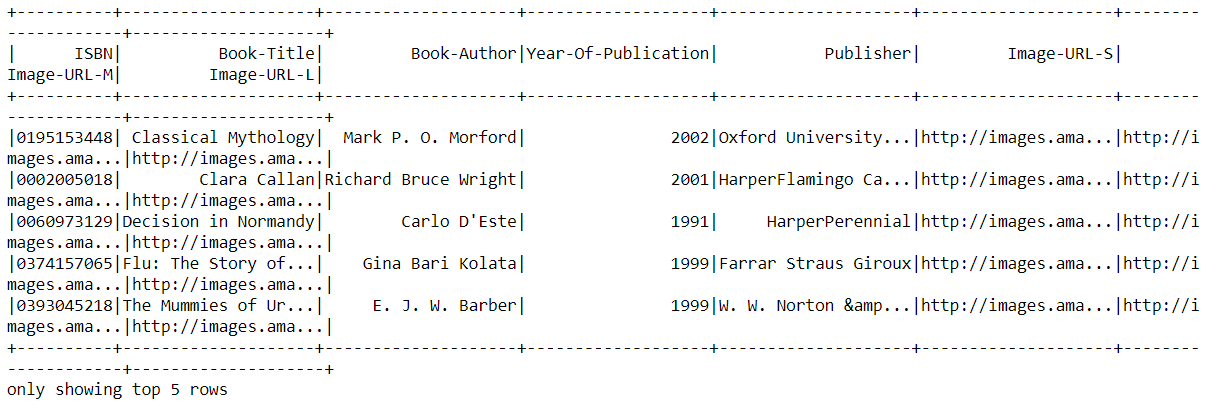
#Read books csv to load book title

books\_csv = spark.read.option("header","true").option("quote", "\"").option("delimiter", ";").csv("s3://ai-in-aws/awsglue-datasets/BX-Books.csv")

We will explore if the data has been read correctly

#Explore the first few records of the books\_csv dataframe

books\_csv.show(5)



We will once again clean up ISBN column

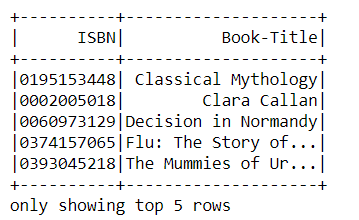
#Clean up the ISBN column

books\_csv = books\_csv.withColumn("ISBN", F.lower(F.trim(F.col("ISBN"))))

#Drop columns that are not of interest

books\_csv = books\_csv.select('ISBN', 'Book-Title')

books\_csv.show(5)



We will now obtain book titles for those books that met our threshold (rated by at least 6 users)

fil\_books = fil\_books.join(books\_csv, on=['ISBN'], how='inner')\

.select(F.col("ISBN"),

F.col("count"),

F.col("Book-Title")

)

#Let's inspect the number of books with title

fil\_books.count()

32,978

**Select relevant book ratings**

Ratings dataset is now filtered to only retain information on users and books that met the thresholds defined earlier.

# Create filtered ratings containing user and book indexes, along with rating

fil\_ratings = ratings.join(fil\_users, on=['User-ID'], how='inner').join(fil\_books, on=['ISBN'], how='inner')\

.select(F.col("ISBN"),

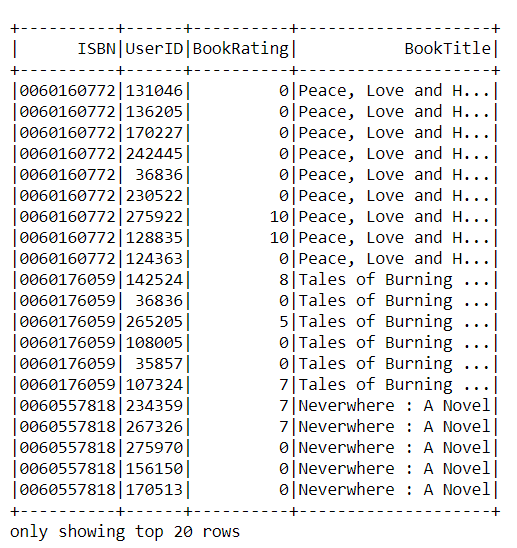
F.col("User-ID").alias("UserID"),

F.col("Book-Rating").alias("BookRating"),

F.col("Book-Title").alias("BookTitle")

)

fil\_ratings.show()



fil\_ratings.select('ISBN').distinct().count()

32745

**Create integer indexes for users and books**

Unique integer indexes for both users and books are required by Object2Vec, which is a built-in SageMaker algorithm used to predict affinity of a user towards a book (recommender), to fit the model to the data

#Determine unique users and books from ratings

uniq\_users = value\_counts(fil\_ratings, 'UserID')

uniq\_books = value\_counts(fil\_ratings, 'ISBN')

w1 = Window.orderBy("UserID")

uniq\_users = uniq\_users.withColumn("user\_ind", F.row\_number().over(w1)-1)

w2 = Window.orderBy("ISBN")

uniq\_books = uniq\_books.withColumn("book\_ind", F.row\_number().over(w2)-1)

#Check the indexes created

row1 = uniq\_books.agg({"book\_ind": "max"}).collect()[0]

print(row1)

Row(max(book\_ind)=32744)

row2 = uniq\_users.agg({"user\_ind": "max"}).collect()[0]

print(row2)

Row(max(user\_ind)=10298)

Add indexes (both user and book) to the filtered ratings dataset

# Create filtered ratings containing user and book indexes, along with rating

upd\_fil\_ratings = fil\_ratings.join(uniq\_users, on=['UserID'], how='inner').join(uniq\_books, on=['ISBN'], how='inner')\

.select(F.col("ISBN"),

F.col("UserID"),

F.col("BookRating"),

F.col("BookTitle"),

F.col("book\_ind"),

F.col("user\_ind"))

upd\_fil\_ratings.count()

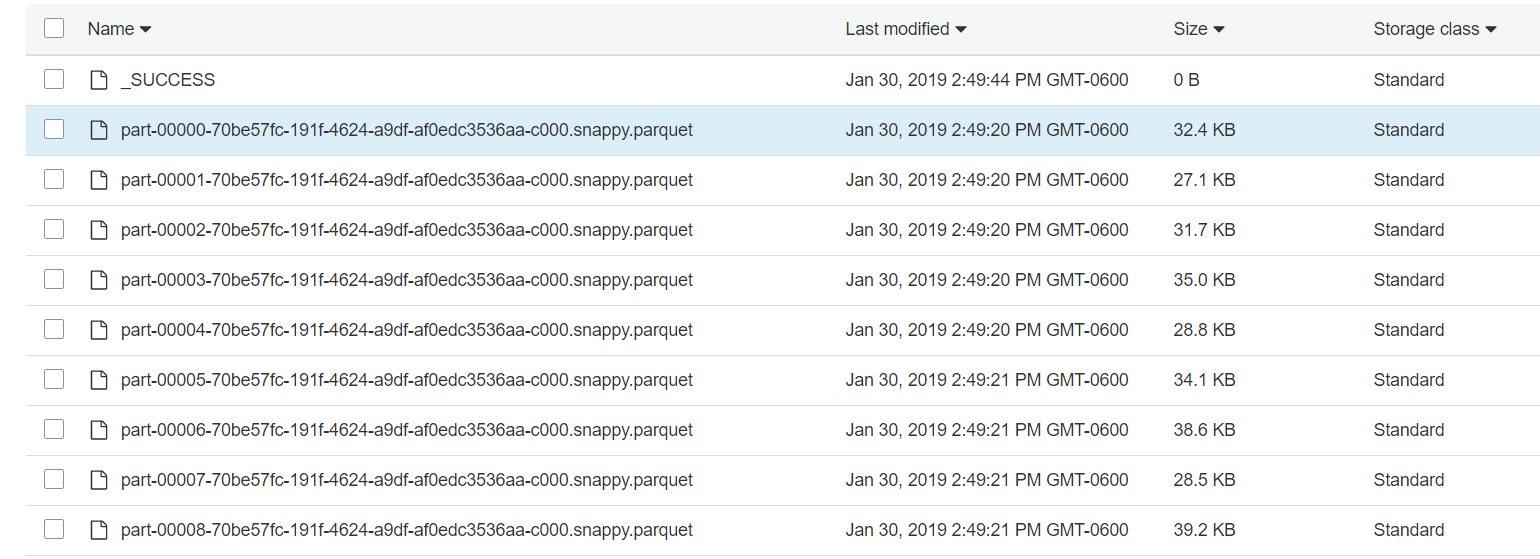
520330

**Save the final book ratings information to S3 bucket**

output\_loc = s3\_bucket + output\_prefix

upd\_fil\_ratings.write.parquet(output\_loc+"/bookratings.parquet", mode='overwrite')

Since the data is parallel processed on EMR cluster, the output contains several parquet files. Apache Parquet is an open source compressed columnar storage format in the Apache Hadoop ecosystem. Compared to the traditional approach where data is stored in row-oriented approach, parquet allows us to be more efficient in terms of storage and performance.



Stop the notebook and terminate the cluster after you are done storing the processed dataset in S3 to avoid unnecessary costs.

**Conduct training in Amazon SageMaker**

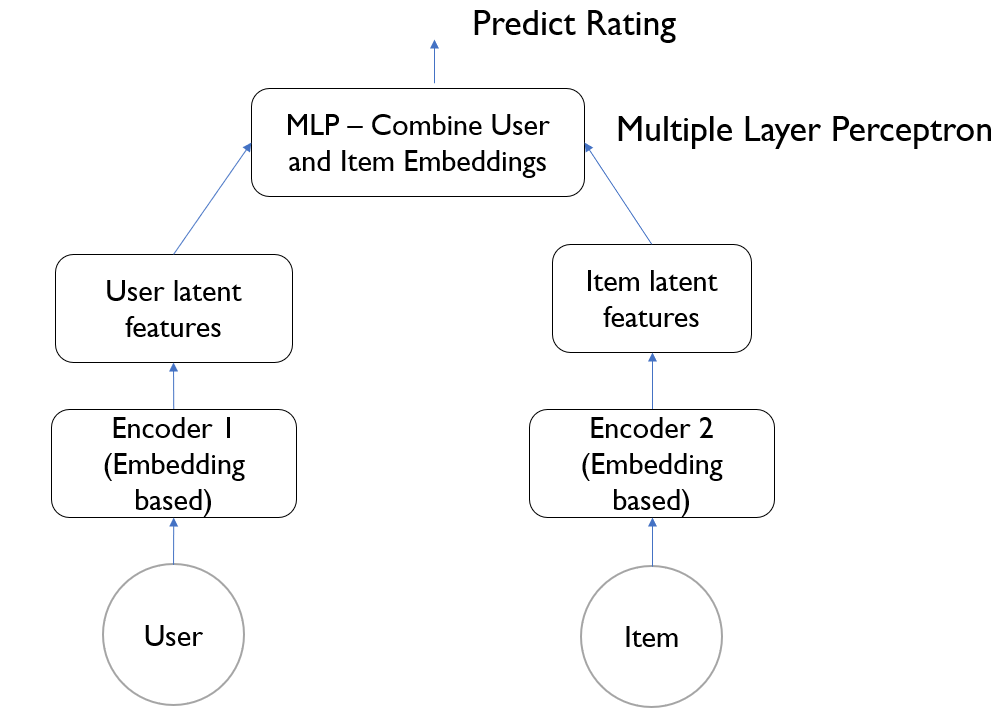
Now that we have the dataset ready, it is time to start transforming it into a format required by ObjectToVec algorithm.

But before that, let’s spend a few minutes understanding how the algorithm works. It is multi-purpose algorithm that can create lower dimensional embeddings of higher dimensional objects. This process is known as dimensionality reduction, most commonly implemented through a statistical procedure called Principal Component Analysis (PCA). However, Object2Vec uses neural networks to learn these embeddings. Some of the common applications of these embeddings include customer segmentation and product search. In the case of customer segmentation, similar customers appear closer in the lower dimensional space. A customer can be defined through multiple attributes – Name, age, home address, email address. With regards to product search, because product embeddings capture semantics of the underlying data, any combination of search terms can be used to retrieve target product. The embedding of these search terms (i.e. the semantics) should just match that of the product.

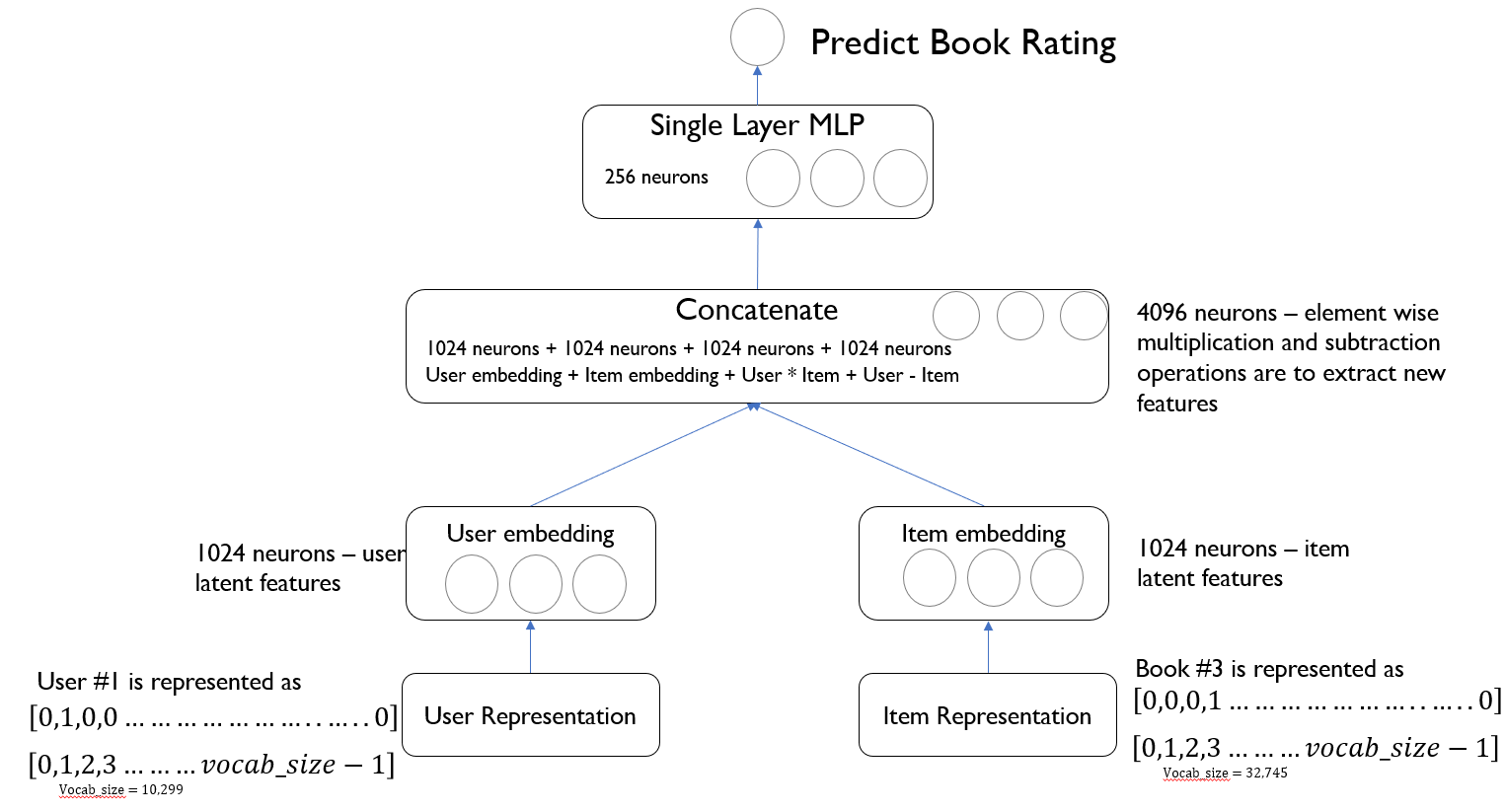
*Collaborative Recommendation System:*

Additionally, Object2vec can learn embeddings of pairs of objects. In our case, the higher the rating of the book, the stronger the relationship between the user and the book. The idea is that users with similar tastes are likely to rate similar books higher. Object2vec approximates the book rating by using embeddings of users and books. The closer a user is to some books, the higher is the rating given by that user to the books.

We provide the algorithm with (user\_ind and book\_ind) pairs; for each such pair, we also provide a “label” that tells the algorithm whether the user and book are similar or not. The “label” in our case is book rating. Therefore, the trained model can be used to predict rating of a book for a given user – the book, in this case, has never been rated by the user.



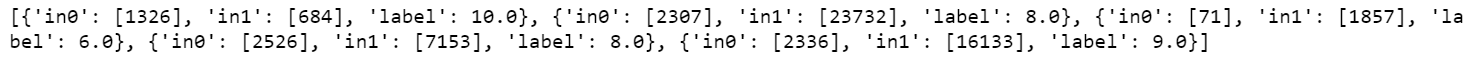
Below is the network architecture of the best performing Object2Vec model:



Object2vec starts with representing user and book with one hot encoding. To explain, in our case, a user can be represented with an array of size 10,299 – there are a total of 10,299 unique users in the dataset. User #1 can be represented with denoting 1 at position 1, while all the other positions in the array has zeros. Books can also be represented in a comparable manner. It is time to now reduce dimensionality of these representations. Therefore, the algorithm uses an embedding layer with 1024 neurons, each for an user and a book. Object2vec further extracts additional features by conducting element-wise multiplication and subtraction between 1024 user embedding neurons and 1024 item embedding neurons. In other words, the user and book embeddings are compared in different ways. Overall, we will then have 4096 neurons when all the neurons from the previous layers are merged. The algorithm then uses single perceptron layer with 256 neurons. This perceptron layer is then fully connected to output layer with one neuron. This one neuron will then predict rating of a book given by a user.

Now that we understand how the algorithm works, let’s dive into the training process:

1. Data Processing – feed data in the form of JSON lines; random shuffle the data for optimal performance. As you will see below, we send data in the format – user index, book index, label=rating



1. Model Training – we pass both training and validation data to the algorithm. There are multiple hyperparameters that we can configure to fine tune the model performance. We will review them in the following sections. The objective function is our case is to minimize MSE (mean squared error). Error is the difference between label (actual value) and predicted rating.
2. Inference – once the model is trained, we will deploy it as an endpoint. We will then pass pairs of user and book indexes. We will compare model performance with baseline metrics.

Let’s start by importing the relevant python modules

#Install relevant python packages

!pip install --upgrade pip

!pip install jsonlines

!pip install pyarrow

#Import relevant modules

import os

import io

import string

import sys

import csv, jsonlines

import numpy as np

import pandas as pd

import copy

import random

import boto3

from sagemaker.session import s3\_input

import sagemaker

from sagemaker import get\_execution\_role

from sagemaker.amazon.amazon\_estimator import get\_image\_uri

import s3fs

import pyarrow.parquet as pq

%matplotlib inline

import matplotlib.pyplot as plt

**Read the dataset from s3**

We will read the parquet files from S3 bucket

### Read the dataset from s3

#Read the prepared dataset from s3 bucket

s3 = s3fs.S3FileSystem()

s3\_bucket = 's3://ai-in-aws/'

input\_prefix = 'object2vec/bookratings/bookratings.parquet'

dataset\_name = s3\_bucket + input\_prefix

df\_bkRatngs = pq.ParquetDataset(dataset\_name, filesystem=s3).read\_pandas().to\_pandas()

df\_bkRatngs['BookRating'] = pd.to\_numeric(df\_bkRatngs['BookRating'])

**Remove outliers in the dataset**

There are considerable number of books with zero ratings. These ratings make the dataset highly imbalanced. We will therefore remove zero rating entries

#Inspect the book ratings dataset

#Remove zero ratings from the dataset since there is a large number of them

df\_bkRatngs = df\_bkRatngs[df\_bkRatngs.BookRating > 0]

df\_bkRatngs.head()

len(df\_bkRatngs)

169555

We will determine the number of unique users and books in the final ratings dataset

# Number of unique users

# Number of unique books

#10467 (users), 32,977 (books)

num\_users = len(df\_bkRatngs['user\_ind'].unique())

num\_books = len(df\_bkRatngs['book\_ind'].unique())

print('Number of unique users: ', num\_users)

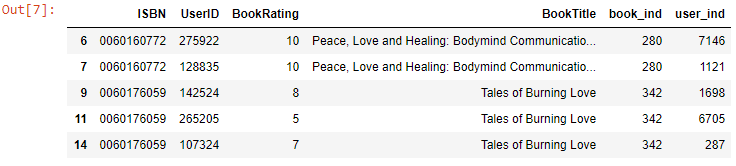
print('Number of unique books: ', num\_books)

Number of unique users: 9648

Number of unique books: 30036

We will examine the dataset to ensure there are no data quality issues

df\_bkRatngs.head()



We will need to reset the user and book indexes, since we just removed few entries from the ratings dataset

#Reset user and book indexes

uniq\_users = pd.DataFrame(data=df\_bkRatngs['UserID'].unique(), columns=['UserID'])

uniq\_users['user\_ind'] = np.arange(num\_users)

uniq\_books = pd.DataFrame(data=df\_bkRatngs['ISBN'].unique(), columns=['ISBN'])

uniq\_books['book\_ind'] = np.arange(num\_books)

Create final ratings dataset, with new indexes

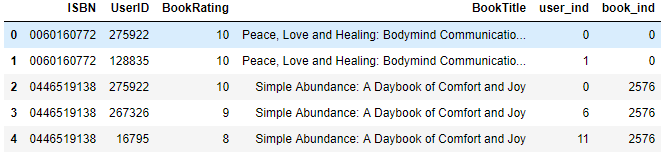
#Create final ratings dataset

#pd.merge(restaurant\_ids\_dataframe, restaurant\_review\_frame, on='business\_id', how='outer')

df\_bkRatngs = df\_bkRatngs.drop(columns = ['user\_ind', 'book\_ind'])

df\_bkRatngs = pd.merge(df\_bkRatngs, uniq\_users, on='UserID', how='inner').merge(uniq\_books, on='ISBN', how='inner')

df\_bkRatngs.head()



#Save the clean dataframe

df\_bkRatngs.to\_csv('ClndBookRatings.csv', header='true', index=False)

**Utility Functions**

These are commonly used utility functions that are segmented into a separate python file, utilityfunctions.py

## some utility functions

import numpy as np

import csv, jsonlines

import os

import io

import string

import sys

import pandas as pd

# Output data in the format required by object2vec

def load\_df\_data(df, verbose=True):

"""

input: a dataframe that

has format users - books - ratings - etc

output: a list, where each row of the list is of the form

{'in0':userID, 'in1':bookID, 'label':rating}

"""

to\_data\_list = list()

users = list()

items = list()

ratings = list()

userIDMap = list()

unique\_users = set()

unique\_items = set()

for idx, row in df.iterrows():

to\_data\_list.append({'in0':[int(row['user\_ind'])], 'in1':[int(row['book\_ind'])], 'label':float(row['BookRating'])})

users.append(row['user\_ind'])

items.append(row['book\_ind'])

ratings.append(float(row['BookRating']))

unique\_users.add(row['user\_ind'])

unique\_items.add(row['book\_ind'])

if verbose:

print("There are {} ratings".format(len(ratings)))

print("The ratings have mean: {}, median: {}, and variance: {}".format(

round(np.mean(ratings), 2),

round(np.median(ratings), 2),

round(np.var(ratings), 2)))

print("There are {} unique users and {} unique books".format(len(unique\_users), len(unique\_items)))

return to\_data\_list, ratings

# Save jsonlines to a file

def write\_data\_list\_to\_jsonl(data\_list, to\_fname):

"""

Input: a data list, where each row of the list is a Python dictionary taking form

{'in0':userID, 'in1':bookID, 'label':rating}

Output: save the list as a jsonline file

"""

with jsonlines.open(to\_fname, mode='w') as writer:

for row in data\_list:

#print(row)

writer.write({'in0':row['in0'], 'in1':row['in1'], 'label':row['label']})

print("Created {} jsonline file".format(to\_fname))

#Transform test data in the format required by object2vec

def data\_list\_to\_inference\_format(data\_list, binarize=True, label\_thres=3):

"""

Input: a data list

Output: test data and label, acceptable by SageMaker for inference

"""

data\_ = [({"in0":row['in0'], 'in1':row['in1']}, row['label']) for row in data\_list]

print("data\_ :", data\_)

data, label = zip(\*data\_)

print("data :", data)

print("label :", label)

infer\_data = {"instances":data}

print("infer\_data : ", infer\_data)

if binarize:

label = get\_binarized\_label(list(label), label\_thres)

return infer\_data, label

# Compute Mean Squared Error for model evaluation

def get\_mse\_loss(res, labels):

if type(res) is dict:

res = res['predictions']

assert len(res)==len(labels), 'result and label length mismatch!'

loss = 0

for row, label in zip(res, labels):

if type(row)is dict:

loss += (row['scores'][0]-label)\*\*2

else:

loss += (row-label)\*\*2

return round(loss/float(len(labels)), 2)

# Create user and books dictionary

# User dictionary: users[userID] : {bookID, rating}

# Book dictionary: books[bookID] : {userID1, userID2..}

def jsnl\_to\_augmented\_data\_dict(jsnlRatings):

"""

Input: json lines that

has format users - books - ratings - etc

Output:

Users dictionary: keys as user ID's; each key corresponds to a list of book ratings by that user

Books dictionary: keys as book ID's; each key corresponds a list of ratings of that book by different users

"""

to\_users\_dict = dict()

to\_books\_dict = dict()

for row in jsnlRatings:

if row['in0'][0] not in to\_users\_dict:

to\_users\_dict[row['in0'][0]] = [(row['in1'][0], row['label'])]

else:

to\_users\_dict[row['in0'][0]].append((row['in1'][0], row['label']))

if row['in1'][0] not in to\_books\_dict:

to\_books\_dict[row['in1'][0]] = list(row['in0'])

else:

to\_books\_dict[row['in1'][0]].append(row['in0'])

return to\_users\_dict, to\_books\_dict

Load the ratings dataframe as json lines

#Load the data as json lines (the format in which object2vec requires input)

data\_list, ratings\_list = load\_df\_data(df\_bkRatngs)



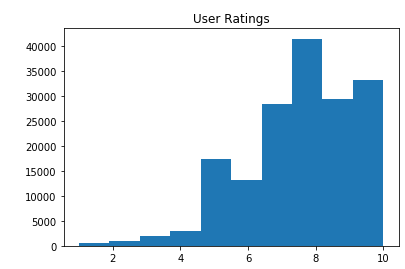
**Look at Ratings distribution**

# Distribution of rating

f = plt.figure(1)

plt.hist(ratings\_list)

plt.title("User Ratings")



**Create Training, Validation and Test datasets**

#data\_list, ratings\_list = load\_df\_data(df\_bkRatngs)

random.shuffle(data\_list)

n\_train = int(0.8 \* len(data\_list))

# split train and test

train\_list = data\_list[:n\_train]

test\_list = data\_list[n\_train:]

print(len(train\_list))

print(len(test\_list))

# further split test set into validation set and test set

n\_test = len(test\_list)

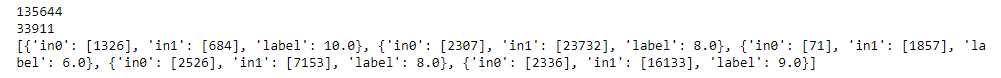
val\_list = test\_list[:n\_test//2]

test\_list = test\_list[n\_test//2:]

random.shuffle(train\_list)

random.shuffle(val\_list)

print(train\_list[:5])



**Writing data in the format required by Object2Vec**

The object2vec algorithm is used predict exact ratings of a book by a given user. We provide the algorithm userID, itemID pairs. For each such pair, we also provide "label" that tells the algorithm how strongly user and item are associated

We will create json lines for both training and validation datasets

write\_data\_list\_to\_jsonl(copy.deepcopy(train\_list), 'train\_r.jsonl')

write\_data\_list\_to\_jsonl(copy.deepcopy(val\_list), 'validation\_r.jsonl')

**Model Training**

Define S3 bucket that hosts the data and model, and upload data to the relevant bucket. Define the paths (on s3) containing training and validation data

bucket = 'ai-in-aws' #Designate your own bucket name

input\_prefix = 'object2vec/bookratings/input'

output\_prefix = 'object2vec/bookratings/output'

s3\_client = boto3.client('s3')

input\_paths = {} #initialize dictionary

output\_path = os.path.join('s3://', bucket, output\_prefix)

for data\_name in ['train', 'validation']:

pre\_key = os.path.join(input\_prefix, 'rating', f'{data\_name}') #F-string: embed python expression inside string literal

fname = '{}\_r.jsonl'.format(data\_name)

data\_path = os.path.join('s3://', bucket, pre\_key, fname)

#upload data to S3

s3\_client.upload\_file(fname, bucket, os.path.join(pre\_key, fname))

#Create definition of input data; data\_path - path containing s3 data

input\_paths[data\_name] = s3\_input(data\_path, distribution='ShardedByS3Key', content\_type='application/jsonlines')

print('Uploaded {} data to {} and defined input path'.format(data\_name, data\_path))

print('Trained model will be saved at', output\_path)

**Get Docker image of ObjectToVec algorithm**

sess = sagemaker.Session()

role = get\_execution\_role()

#Get docker image for the ObjectToVec algorithm

container = get\_image\_uri(boto3.Session().region\_name, 'object2vec') #input: region name, name of the algorithm

**Let’s define hyperparameters**

static\_hyperparameters = {

"\_kvstore": "device", #type of GPU

"\_num\_gpus": "auto",

"\_num\_kv\_servers": "auto",

#"dropout": 0.3,

"bucket\_width": 0,

"early\_stopping\_patience": 2,

"early\_stopping\_tolerance": 0.01,

#"enc0\_cnn\_filter\_width": 3, #3X3 square

"enc0\_layers": "auto",

"enc0\_max\_seq\_len": 1,

"enc0\_network": "pooled\_embedding", # average pooling

"enc0\_token\_embedding\_dim": 300,

"enc0\_vocab\_size": 10299, #number of unique users

"enc1\_layers": "auto",

….

"enc1\_max\_seq\_len": 1,

"enc1\_network": "pooled\_embedding",

"enc1\_token\_embedding\_dim": 300,

"enc1\_vocab\_size": 32745, #number of unique books

"enc\_dim": 1024,

"epochs": 10,

#"learning\_rate": 0.01,

"mini\_batch\_size": 64,

"mlp\_activation": "relu",

"mlp\_dim": 256,

"mlp\_layers": 1,

"optimizer": "adam",

"output\_layer": "mean\_squared\_error"

}

## get estimator

regressor = sagemaker.estimator.Estimator(container,

role,

train\_instance\_count=1,

train\_instance\_type='ml.p2.xlarge',

output\_path=output\_path,

sagemaker\_session=sess)

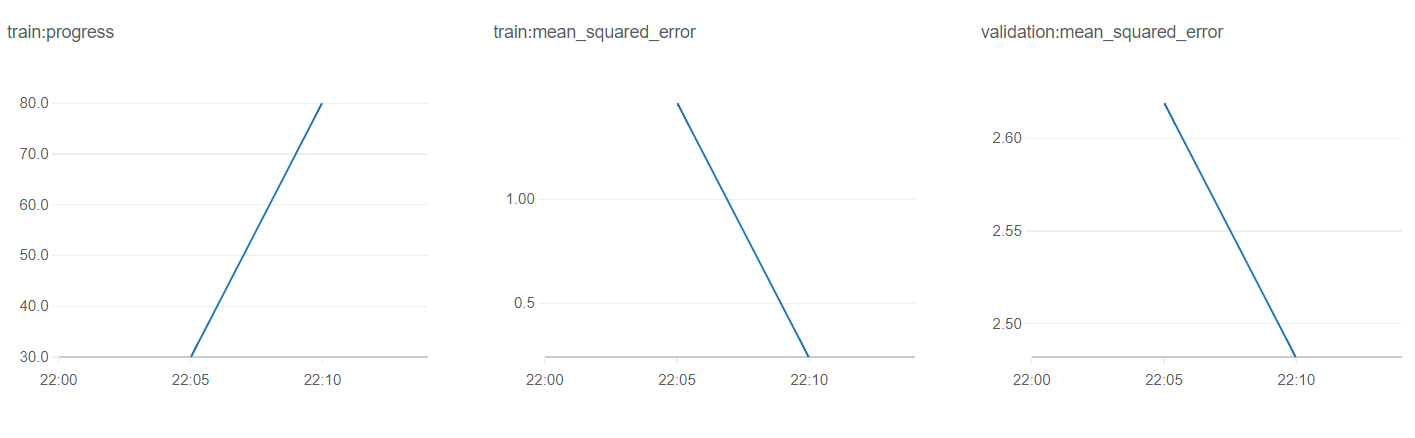
## set hyperparameters

regressor.set\_hyperparameters(\*\*static\_hyperparameters)

## train, tune, and test the model

regressor.fit(input\_paths)

To monitor the training job in progress, navigate to *Training* section on the left hand side. Click on *Training Jobs* and then on the job name of your current job and then navigate to the monitor section. As you can see, as the training MSE decreases, the validation MSE also decreases.



**Model Deployment**

SageMaker SDK offers methods to seamlessly deploy the trained model. First, we create the model using create\_model method of the SageMaker estimator object *regressor*. Once the model is created, it is deployed as an endpoint via deploy method. All you need to specify is number of instances you want to launch and the type of them.

from sagemaker.predictor import json\_serializer, json\_deserializer

# create a model using the trained algorithm

regression\_model = regressor.create\_model(

serializer=json\_serializer,

deserializer=json\_deserializer,

content\_type='application/json')

# deploy the model

predictor = regression\_model.deploy(initial\_instance\_count=1, instance\_type='ml.m4.xlarge')

**Model Evaluation**

Convert the data in the format required by Object2vec

test\_data, test\_label = data\_list\_to\_inference\_format(copy.deepcopy(test\_list), binarize=False)

# Send data to the endpoint to get predictions

prediction = predictor.predict(test\_data)

print("The mean squared error on test set is %.3f" %get\_mse\_loss(prediction, test\_label))

The mean squared error on test set is 2.390

**Baseline 1**

Global (across all users) Average book rating from training dataset. Compare this average with ratings from test dataset (actual value)

#Option 1

train\_label = [row['label'] for row in copy.deepcopy(train\_list)]

bs1\_prediction = round(np.mean(train\_label), 2)

print("The mse loss of the Baseline 1 is {}".format(

get\_mse\_loss(len(test\_label)\*[bs1\_prediction], test\_label)))

The mse loss of the Baseline 1 is 3.15

#Create user dictionary for users in training dataset

to\_users\_dict, to\_books\_dict = jsnl\_to\_augmented\_data\_dict(train\_list)

**Baseline 2**

Instead of global average book rating, let's use average book rating by user. To illustrate, for a given user, we will take average book rating across all the books s/he rated. We will use this rating as our prediction and compare it to the label (actual value).

#Determine how many test users are in training dataset

#We can only compute average book rating by user, only if the user is in training dataset

dftest = pd.DataFrame()

dftrain = pd.DataFrame()

#test

tusrlist = []

for row in test\_list:

tusrlist.append(row['in0'][0])

#train

tr\_usrlist = []

for row in train\_list:

tr\_usrlist.append(row['in0'][0])

dftest = pd.DataFrame({'userID':tusrlist})

dftrain = pd.DataFrame({'userID':tr\_usrlist})

dfmatch = dftest[dftest['userID'].isin(dftrain['userID'])]

len(dftest['userID'].unique())

#5,631 users are in test dataset

len(dftrain['userID'].unique())

#9,489 users are in train dataset

len(dfmatch['userID'].unique())

# Out of 5,631 users in test dataset, 5,541 users only are in training set

#Option 2:

def bs2\_predictor(test\_data, user\_dict):

test\_data = copy.deepcopy(test\_data['instances'])

predictions = list()

for row in test\_data:

userID = int(row["in0"][0])

# predict book rating based on local average of user's prediction

if userID in user\_dict:

local\_books, local\_ratings = zip(\*user\_dict[userID])

local\_ratings = [float(score) for score in local\_ratings]

predictions.append(np.mean(local\_ratings))

else:

#For users not in training set, let's user global average rating

predictions.append(bs1\_prediction)

#print ("predictions per user", predictions[-1])

return predictions

bs2\_prediction = bs2\_predictor(test\_data, to\_users\_dict)

print("The mse loss of the Baseline 2 (user-based rating average) is {}".format(

get\_mse\_loss(bs2\_prediction, test\_label)))

The mse loss of the Baseline 2 (user-based rating average) is 2.57

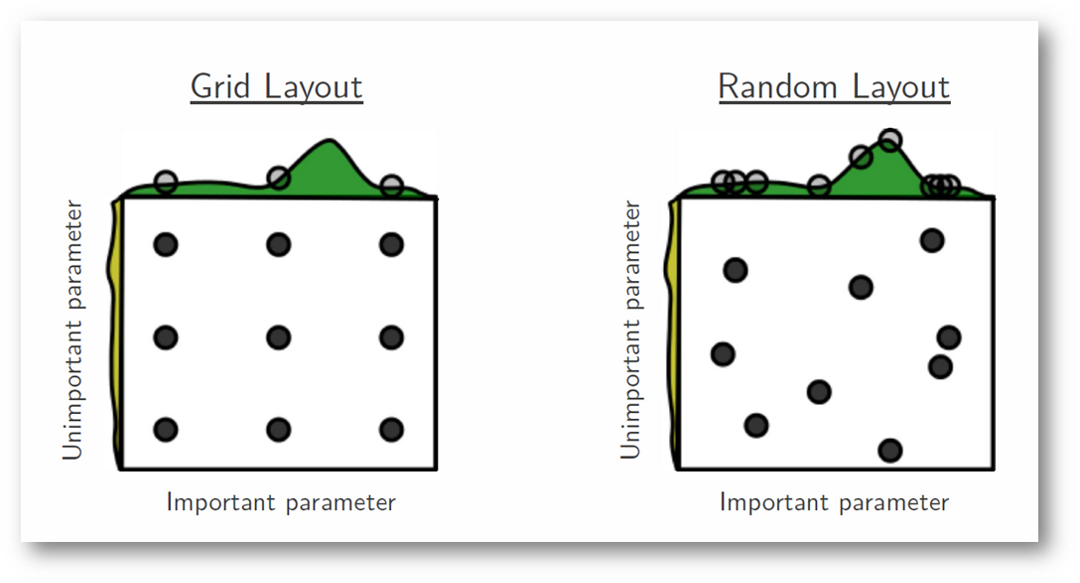
MSE of 2.39 is better than the baselines:

Option 1 MSE: 3.15, where predicted book rating is global average book rating across all users

Option 2 MSE: 2.57, where predicted book rating is average book rating by user

**Run hyperparameter optimization (hpo)**

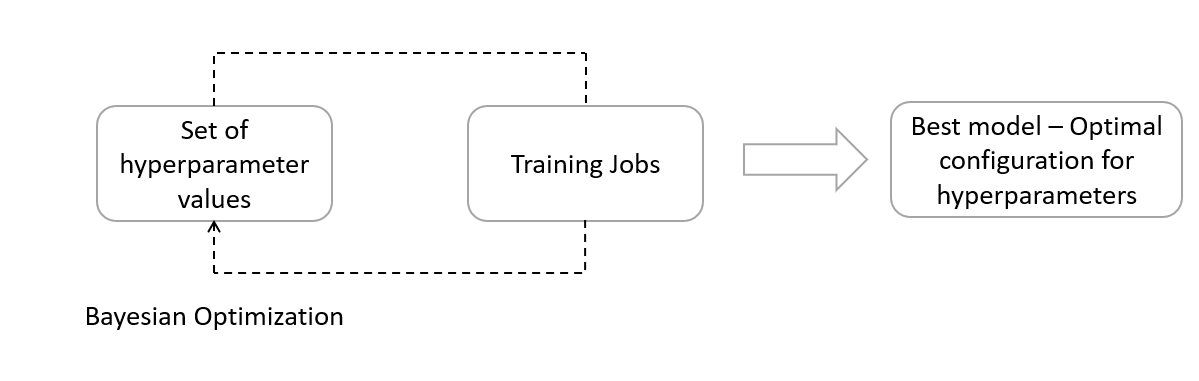
It takes data scientists numerous hours and experiments to arrive at an optimal set of hyperparameters required for best model performance. This process is mostly based on trial and error. Although GridSearch is one of the techniques traditionally used by data scientists, it suffers from curse of dimensionality problem. For example, if we have two hyperparameters, with each taking 5 possible values, we’re looking at calculating objective function 25 times (5x5). As the number of hyperparameters grows, the number of times that the objective function is computed blows out of proportion. Random Search addresses this issue by randomly selecting values of hyperparameters, without doing an exhaustive search of every single combination of hyperparameters. This [paper](http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf) by Bergstra et al. claims that a random search of the parameter space is guaranteed to be more effective than grid search. The idea is that some parameters have much less effect than others on the objective function. This can be reflected in the number of values picked for each parameter in the grid search. Random Search enables exploration of more values for each parameter, given a number of trials.



*Source: Random Search for Hyper-Parameter Optimization (*[*link*](http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf)*)*

Neither of these techniques automate the process of hyperparameter optimization.

Hyperparameter Optimization (HPO) from SageMaker automates the process of selecting optimal combination of hyperparameters. Here is the how the tool works:



Hyperparameter Optimization (HPO) uses bayesian technique to iteratively select combination of hyperparameters to train the algorithm with. To explain, HPO picks the next set of hyperparameters, given the performance of the model and the configuration of hyperparameters in all the historical steps. Also, it employs *acquisition function* to determine where the next best opportunity is to lower the cost function. After a specified number of iterations, you will arrive at an optimal configuration of hyperparameters producing the best model.

For the object2vec algorithm, let’s select the hyperparameters that we want to tune:

* learning\_rate – control the speed with which weights in the neural network are optimized
* dropout - % of the neurons in a layer that are ignored in both forward and backward pass
* enc\_dim – number of neurons to generate user/item embedding
* mlp\_dim – number of neurons in MLP layer
* weight\_decay – factor to prevent overfitting (L2 regularization; causes the weight to decay in proportion to the factor specified)

We will set the objective of HyperparameterTuner, a class in SageMaker, to be to reduce mean squared error for validation dataset. Depending on your budget and time, you can choose how many training jobs you want to run. In this case, I chose to run 10 jobs, with only one job running at a given instance. You can choose to run multiple jobs in parallel.

from time import gmtime, strftime

from sagemaker.tuner import IntegerParameter, CategoricalParameter, ContinuousParameter, HyperparameterTuner

tuning\_job\_name = "object2vec-job-{}".format(strftime("%d-%H-%M-%S", gmtime()))

hyperparameters\_ranges = { "learning\_rate": ContinuousParameter(0.0004, 0.02),

"dropout": ContinuousParameter(0.0, 0.4),

"enc\_dim": IntegerParameter(300, 400),

"mlp\_dim": IntegerParameter(1000, 1500),

"weight\_decay": ContinuousParameter(0, 300) }

objective\_metric\_name = 'validation:mean\_squared\_error'

regressor = sagemaker.estimator.Estimator(container,

role,

train\_instance\_count=1,

train\_instance\_type='ml.p2.xlarge',

output\_path=output\_path,

sagemaker\_session=sess)

## set hyperparameters

regressor.set\_hyperparameters(\*\*static\_hyperparameters)

tuner = HyperparameterTuner(regressor,

objective\_metric\_name,

hyperparameters\_ranges,

objective\_type='Minimize',

max\_jobs=10,

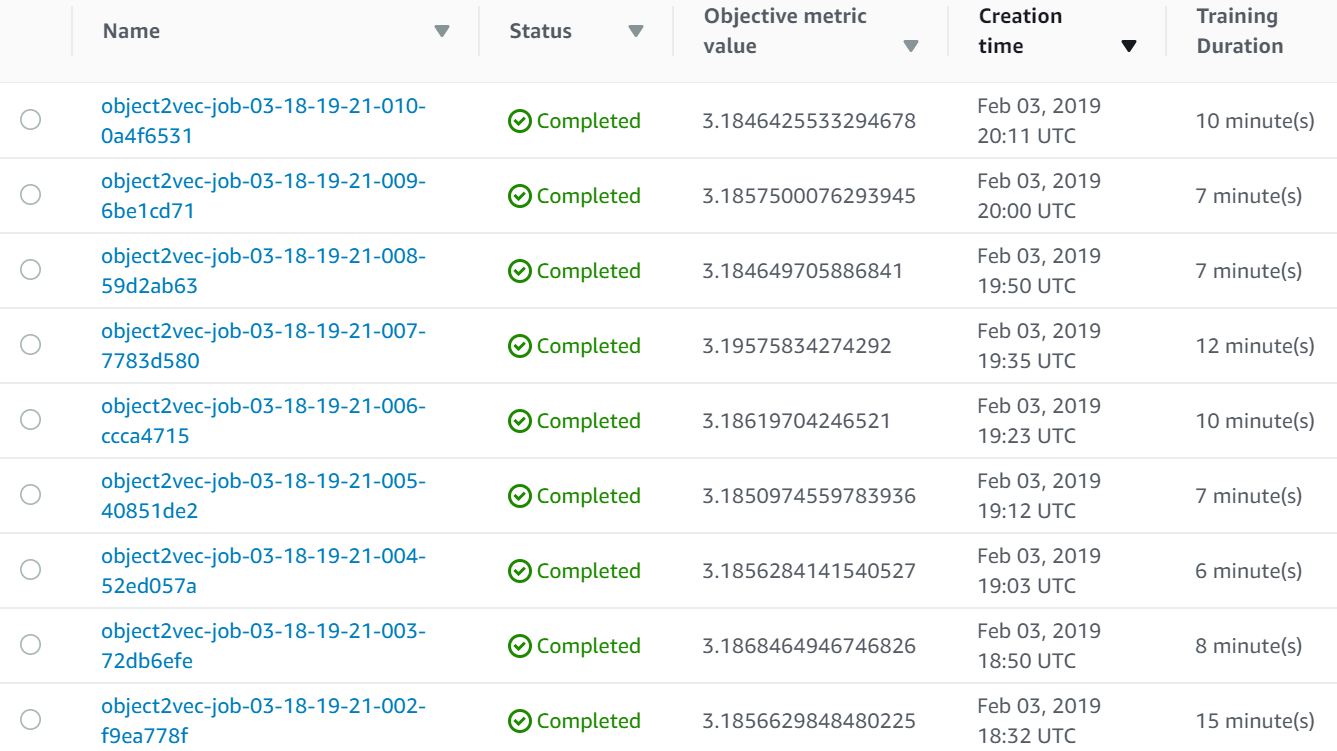
max\_parallel\_jobs=1)

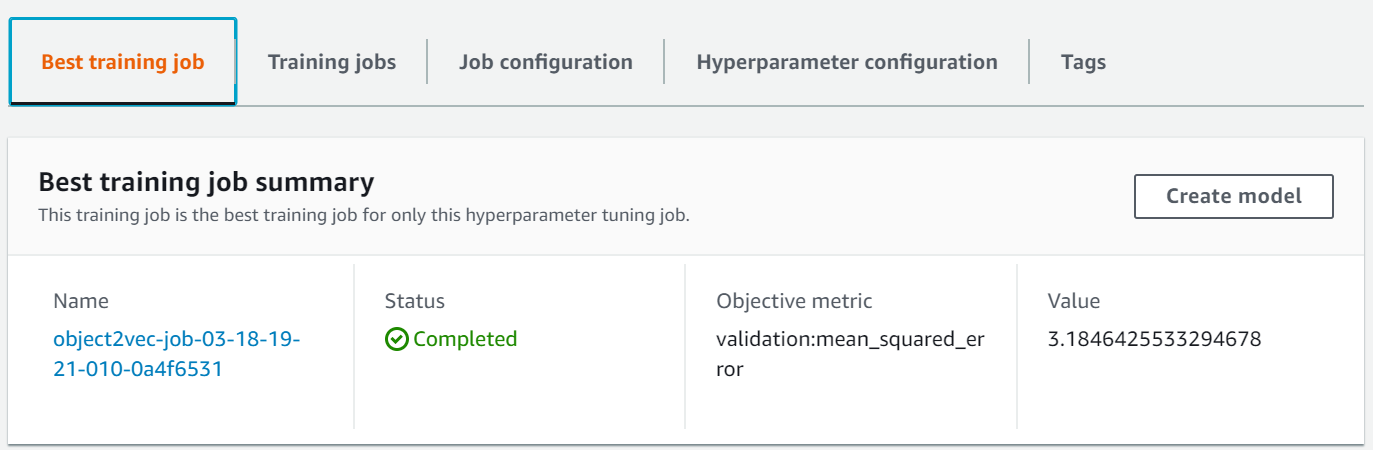
tuner.fit({'train': input\_paths['train'], 'validation': input\_paths['validation']},

job\_name=tuning\_job\_name, include\_cls\_metadata=False)

tuner.wait()

Listed below are the 10 jobs that have executed. With each job, you can look at the hyperparameters used and the value of the objective function. The best job, with lowest MSE, is presented on the “Best job” tab.



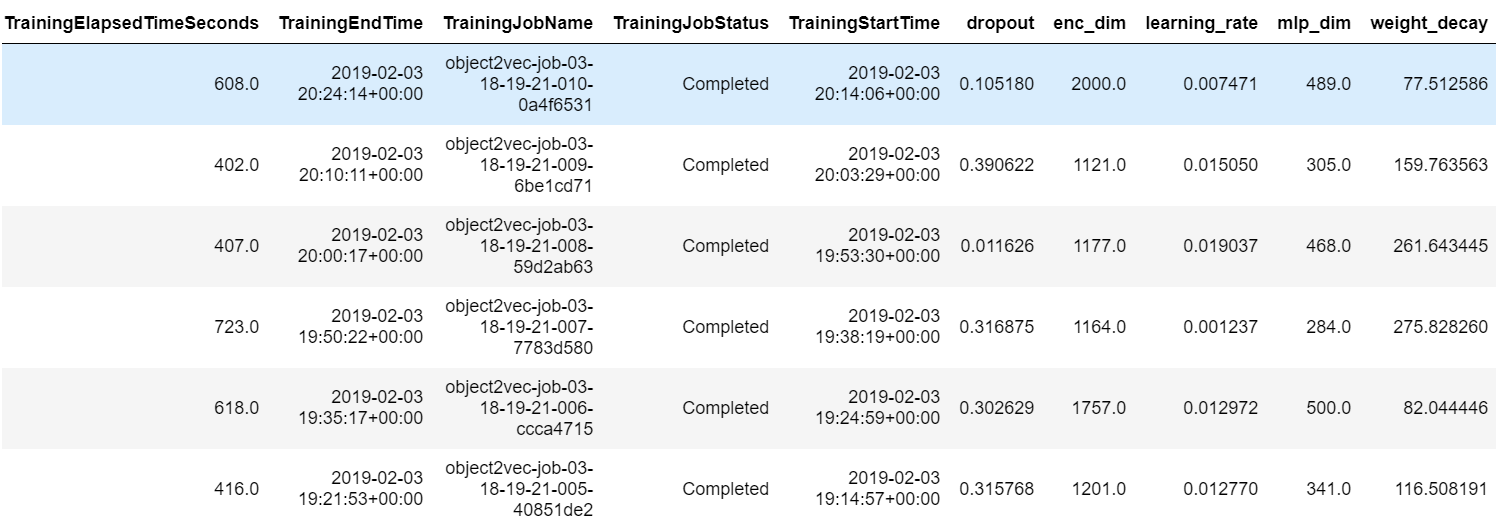


After the jobs are executed, you can run analytics on the results from hyperparameter optimization to answer questions, such as, how did the MSE vary as the tuning jobs are being executed. You can also look at if there is a correlation between MSE and hyperparameters being tuned, such as, learning rate, dropout, weight decay, number of dimensions for both encoder and mlp.

objTunerAnltcs = tuner.analytics()

#Review summary of all the jobs executed

objTunerAnltcs.training\_job\_summaries(force\_refresh=False)



We can look at what combination of hyperparameters produce a model of a certain MSE.

dfTuning = objTunerAnltcs.dataframe(force\_refresh=False)

import bokeh

import bokeh.io

bokeh.io.output\_notebook()

from bokeh.plotting import figure, show

from bokeh.models import HoverTool

# Now plot it

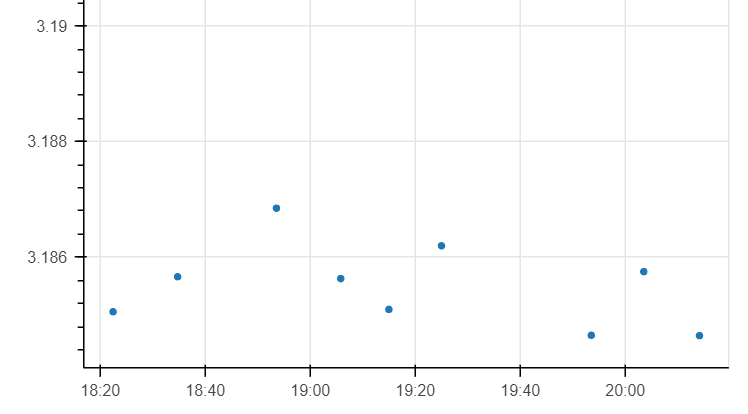
p = figure(plot\_width=500, plot\_height=500,

x\_axis\_type = 'datetime')

p.circle(source=dfTuning, x='TrainingStartTime', y='FinalObjectiveValue')

show(p)

The below plot looks how the MSE changes as the 10 training jobs are being executed. As you can see, the plot is very bumpy. If you increase the number of training jobs, perhaps the hyperperparameter tuning job will converge.

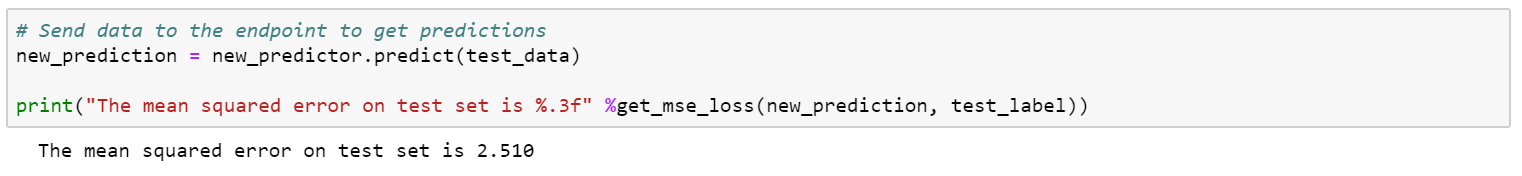


MSE

Time

Based on the hyperparameter tuning jobs and how each of the hyperparameters impacted the objective function, we select a learning rate of .007, mlp dimension of 400, and encoder dimension of 1177. Please refer to the notebook associated with this chapter.

The model has an MSE of 2.510, performing better than all the training jobs launched during hyperparameter tuning. However, this MSE is slightly higher than 2.39 MSE we got from the first iteration of training. If we had reduced the number of hyperparameters being optimized and/or increased the number tuning jobs, the hyper-parameter tuning job may have converged to produce optimized model configuration.



**Experiment Management with SageMaker search**

The goal of experiment management with SageMaker Search is to accelerate the model development and experimentation phase, improving productivity of data scientists and developers, and reduces overall time to market machine learning solutions.

*Machine Learning Lifecycle (continuous experimentation and tuning):* When you initiate training a new learning algorithm, to improve model performance, you conduct hyperparameter tuning. With each iteration of the tuning, you will need to check how the model performance is improving. This leads to hundreds and thousands of experiments and model versions. The whole process slows down the selection of a “final optimized” model. Additionally, it is critical to monitor the performance of a production model. If the predictive performance of the model is degrading, it is important to know how the real-life data is different from the data used during training and validation.

SageMaker Search tackles all the challenges highlighted above by providing the following features:

* Organize, track and evaluate model training experiments
* Seamlessly retrieve the most relevant training runs – runs that can be searched by key attributes – training job name, status, start time, last modified time, failure reason, among other things
* Create leader boards for winning models, cataloguing model training runs – compare models by performance metrics, such as training loss and validation accuracy
* Track the lineage of a deployed model in live environment

Step #1. Conduct multiple model training runs, performing tests with different variations of hyperparameters. In the last section, we have run 10 experiments with different configurations of hyperparameters.

hyperparameters\_ranges = { "learning\_rate": ContinuousParameter(0.0004, 0.02),

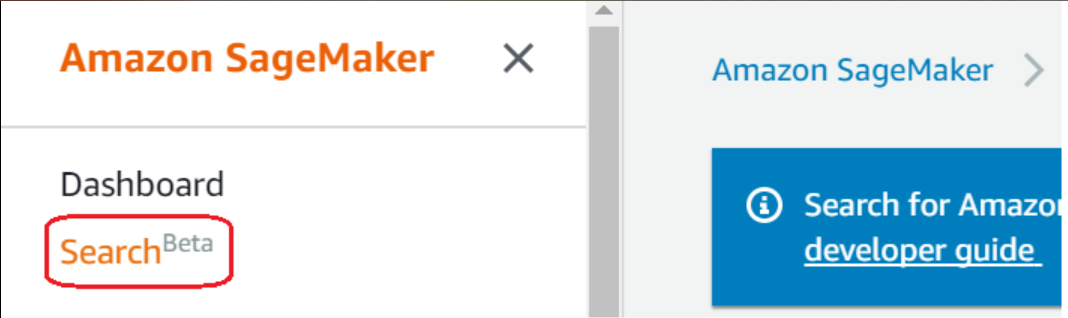
"dropout": ContinuousParameter(0.0, 0.4),

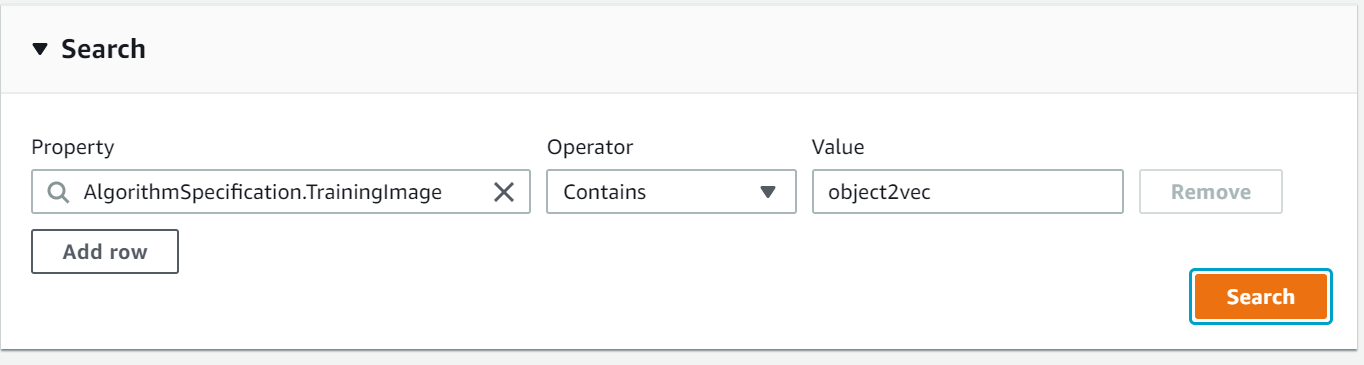
"enc\_dim": IntegerParameter(300, 400),

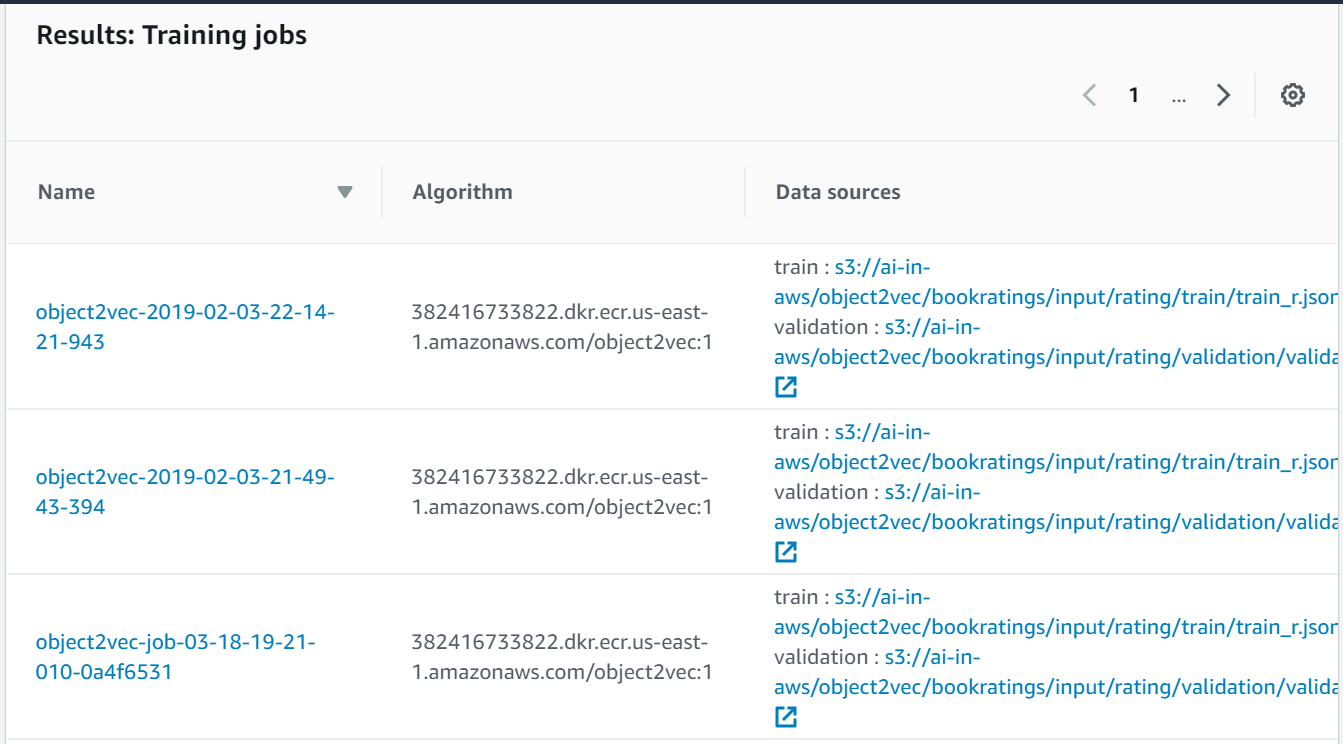
"mlp\_dim": IntegerParameter(1000, 1500),

"weight\_decay": ContinuousParameter(0, 300) }

Step #2: Search and categorize relevant experiments in one place







You can also search and organize experiments using *AWS SDK API for Amazon SageMaker Search*

search\_params={

"MaxResults": 10,

"Resource": "TrainingJob",

"SearchExpression": {

"Filters": [{

"Name": "AlgorithmSpecification.TrainingImage",

"Operator": "Equals",

"Value": "Object2Vec"

}]},

"SortBy": "Metrics.validation:mean\_squared\_error",

"SortOrder": "Descending"

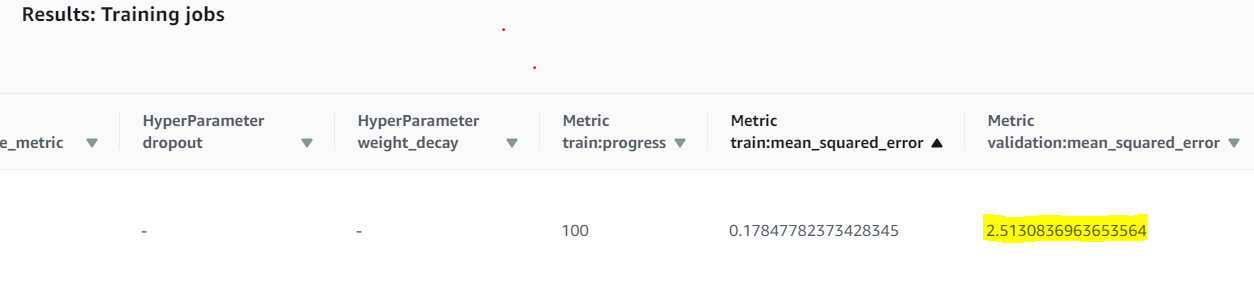
}

sgmclient = boto3.client(service\_name='sagemaker')

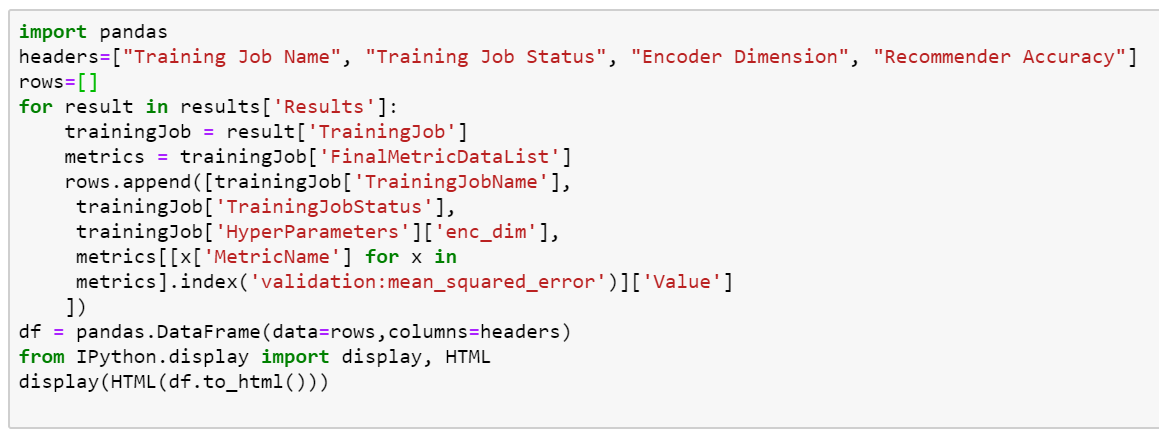
results = sgmclient.search(\*\*search\_params)

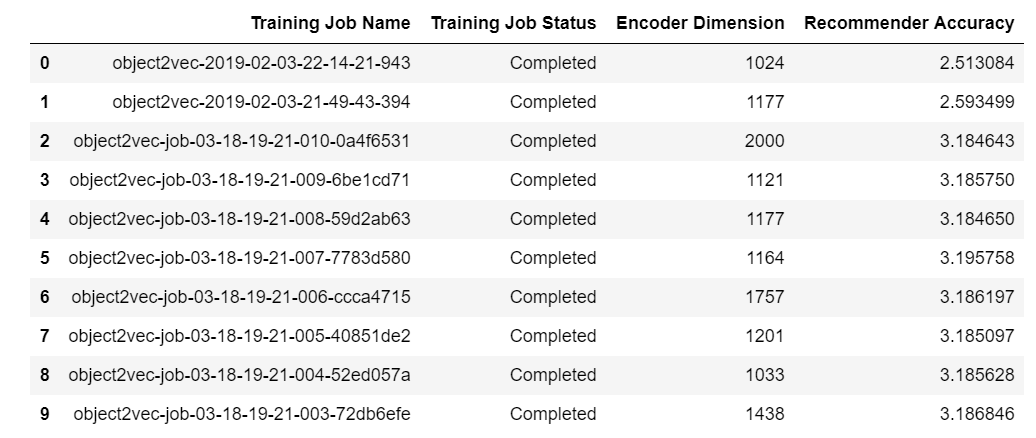
You can search results using any metadata for model training runs – such as learning algorithm, training dataset URI, ranges of numerical values for hyperparameters and model training metrics

Step #3: Sort by objective metric to find the winning model

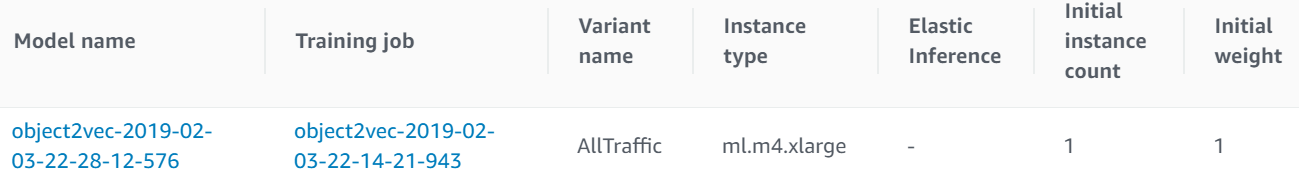


Print leaderboard in Jupyter notebook





Step #4: After finding the winning model, we will deploy the same as an endpoint. We will quickly trace the endpoint to the model training run. Choose **Endpoints** in the left-hand navigation pane and select the endpoint of the winning model. Scroll to the **Endpoint Configuration Settings,** to locate hyperlink to the Model Training Job that was used to create the endpoint



You can use AWS SDK to get endpoint configuration (describe\_endpoint\_config function of sagemaker client). From the configuration, select the model name to retrieve Model Data URL. From the Model Data URL, we will retrieve training job.

**Bring Your Own Model – SageMaker, MXNet, Gluon**

This section focuses on how SageMaker enables you to bring your own deep learning library to the amazon cloud and still utilize the productivity features of SageMaker to automate training and deployment at scale.

The deep learning library we will bring in here is Gluon. Gluon is an open source deep learning library jointly created by AWS and Microsoft. The primary goal of the library is to enable developers to build, train, and deploy machine learning models in cloud.

In the recent past, there has been tremendous amount of research on recommender systems. In particular, Deep Structured Semantic models attempt to capture information from attributes, such as product image, title and description. Extracting semantic information from these additional characteristics will solve the "cold start" problem in the space of recommender systems. In other words, when there is not much consumption history for a given user, a recommender system can propose products similar to the minimal products purchased by the user.

In this section, we will focus on how pre-trained word embeddings can be used in SageMaker to find books similar to the books that a user likes.

We will look at the same book ratings dataset from [book crossing community](http://www2.informatik.uni-freiburg.de/~cziegler/BX/).

We are using conda mxnet p36 kernel. Let’s begin with installing pre-requisites.

#!pip freeze | grep onnx

!pip uninstall onnx

!pip install --upgrade onnx==1.1.1

!pip install --upgrade pip

!pip install mxnet-mkl

!pip install --disable-pip-version-check gluonnlp

!pip install nltk

nltk.download('punkt')

#Import necessary modules

import os

import mxnet as mx

from mxnet import gluon, nd, ndarray

from mxnet.metric import MSE

import pandas as pd

import numpy as np

import sagemaker

from sagemaker.mxnet import MXNet

import boto3

import json

import matplotlib.pyplot as plt

import gluonnlp as nlp

import itertools

from nltk.tokenize import word\_tokenize

import nltk

from sklearn.manifold import TSNE

%matplotlib inline

import matplotlib.pyplot as plt

**Read the book ratings dataset**

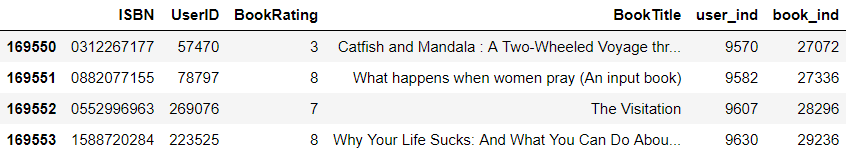
We will read the book ratings dataset we have created in earlier sections.

fn = 'ClndBookRatings.csv'

#print(os.path.isfile(fn))

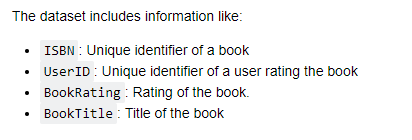
df\_bkRatngs = pd.read\_csv(fn, index\_col=None)

df\_bkRatngs.tail()



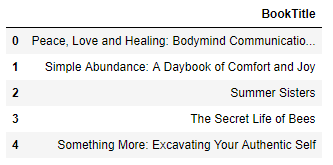
df\_bkRatngs.shape

(169555, 6)



df\_bktitles = pd.DataFrame(df\_bkRatngs['BookTitle'].unique(), columns=['BookTitle'])

df\_bktitles.head()



**Prepare list of words from book titles**

We will now create word tokens from each of the book titles. We remove words that have special characters and/or numbers in them.

# get words from each title, remove words with punctuation marks, numbers, and lower case words

words = []

for i in df\_bktitles['BookTitle']:

tokens = word\_tokenize(i)

#print(tokens)

# remove all tokens that are not alphabetic

words.append([word.lower() for word in tokens if word.isalpha()]) # - this is list of lists

#remove empty strings from words list

words = list(filter(None, words))

#Inspect words list created

len(words)

#words is a list of lists

#print(words)

#print(words[0])

26520

**Get word embedding from pre-trained fastText model**

Now that we have word tokens, we will use fastText to create word embeddings. fastText is a library for learning word embeddings and text classification created by Facebooks's AI Research Lab.

# vocab is a dictionary of key value pairs (value is number of times a word appears in the entire list of titles)

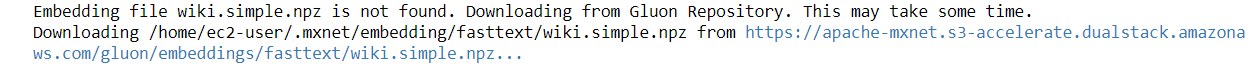
counter = nlp.data.count\_tokens(itertools.chain.from\_iterable(words))

vocab = nlp.Vocab(counter)

# Get the pretrained embeddings for each of the words

fasttext\_simple = nlp.embedding.create('fasttext', source='wiki.simple')

vocab.set\_embedding(fasttext\_simple)



**Create embedding from a book title by averaging across individual word embeddings**

Let’s start by instantiating an empty ndarray (multi-dimensional)

title\_arr\_list = np.empty((0,300), dtype='f')

To create title embedding, average across individual word embeddings for each of the words in the title

for title in words:

#print(title)

#print(vocab.embedding[title])

title\_arr = ndarray.mean(vocab.embedding[title], axis=0, keepdims=True)

#print(title\_arr)

title\_arr\_list = np.append(title\_arr\_list, title\_arr.asnumpy(), axis=0)

title\_arr\_list.shape

(26520, 300)

Let’s look at 532nd title from the list

title\_arr\_list[532]

words[532]

['the', 'doll', 'house', 'sandman', 'book']

**Plot book title embeddings to locate similar books**

In this section, we will plot (t-sne) representation of books to determine which books are semantically similar.

T-distributed stochastic neighborhood embedding (t-SNE) is a machine learning algorithm for visualization. It is a non-linear dimensionality reduction technique

#show tsne plot

#word\_labels = [words]

# find tsne coords for 2 dimensions

tsne = TSNE(n\_components=2, random\_state=0)

np.set\_printoptions(suppress=True)

Y = tsne.fit\_transform(title\_arr\_list)

#select first 20 titles

word\_labels = words[0:20]

x\_coords = Y[0:20, 0]

y\_coords = Y[0:20, 1]

# display scatter plot

plt.figure(figsize=(25, 25))

plt.scatter(x\_coords, y\_coords)

for label, x, y in zip(word\_labels, x\_coords, y\_coords):

#print("label x y", label, x, y)

plt.annotate(label, xy=(x, y), xytext=(0, 0), textcoords='offset points')

#clip\_on=True

plt.xlim(x\_coords.min()+0.00005, x\_coords.max()+0.00005)

plt.ylim(y\_coords.min()+0.00005, y\_coords.max()+0.00005)

plt.show()

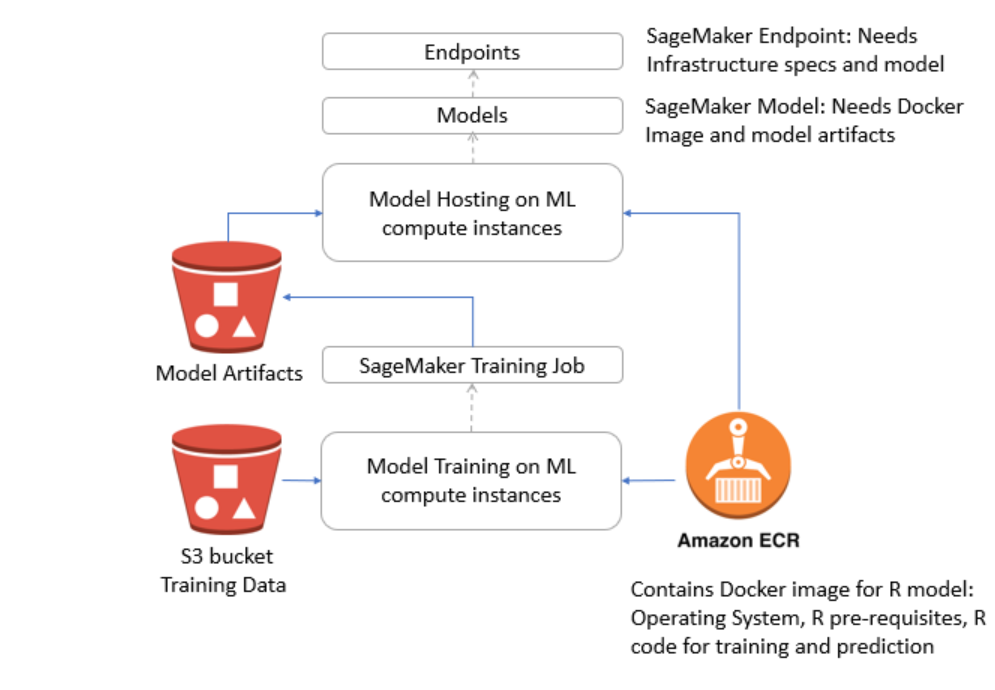
As can be seen below, book titles with semantic similarity are close to each other. For example, titles, such as *anatomy of the spirit – the seven stages of power and healing* and *peace, love and healing – Bodymind communication and the path to an exploration,* which talk about the same subject are located together in the lower dimensional space.



**Bring Your Own Container – R model**

In this section, we will illustrate the process of bringing your own docker container to Amazon SageMaker. Particularly, we will focus on training and hosting R model seamlessly in Amazon SageMaker. Rather than reinventing the wheel on building ML models using SageMaker's built-in algorithms, data scientists and machine learning engineers can re-use their work done in R in SageMaker.

Below is the architecture on how different AWS components interact to train and host R models:



In this section, we will look at the same book ratings dataset from [book crossing community](http://www2.informatik.uni-freiburg.de/~cziegler/BX/). Our goal is to suggest top 5 books to users who are not part of training data set. We will use recommenderlab R package to measure cosine distance between users (User Based Collaborative Filtering). For our target user, we will pick 10 users/neighbors based on the cosine similarity. To estimate top 5 book recommendations for target user, we will compute weighted (cosine similarity) average of ratings for each of the books (in the dataset) based on ratings from selected neighbors. Books with highest weighted ratings are presented to target user.

The first step is to define Dockerfile highlighting requirements to run R code – underlying operating system, R version, R packages and location of R logic for training and inference – and create and publish docker image to EC2 Container Registry (ECR). The second step is to create SageMaker training job, listing training dataset, latest docker image for training and infrastructure specifications. The model artifacts from the training job are stored in relevant s3 bucket. The third step is to host the trained model as an endpoint (REST API). We will use plumber R package to turn R functions into REST endpoints. To host the R model, we will need to create SageMaker model and endpoint. Model artifacts and docker image from training are required to define SageMaker model. Note that the same docker image is leveraged for both training and inference. The SageMaker endpoint takes infrastructure specifications on ML compute instances required to host the newly created model.

R code to train Recommender model: The following R script is run when SageMaker instantiates docker image with “train” command (docker run <image> train)

# Load the required libraries

# Bring in the library to create user book matrix

library(reshape2)

# Bring in the library to compute cosine similarity between users

library(recommenderlab)

# Load the library to host trained R model as a REST endpoint

library(plumber)

library(jsonlite)

# Define location on the container where training data, model and output will be stored

prefix <- '/opt/ml'

input\_path <- paste(prefix, 'input/data', sep='/')

output\_path <- paste(prefix, 'output', sep='/')

model\_path <- paste(prefix, 'model', sep='/')

param\_path <- paste(prefix, 'input/config/hyperparameters.json', sep='/') #similarity method and number of nearest neighbors

# Channel holding training data

channel\_name = 'train'

training\_path <- paste(input\_path, channel\_name, sep='/')

# Define training function

train <- function() {

# Read in hyperparameters

training\_params <- read\_json(param\_path)

# Similarity Method

if (!is.null(training\_params$method)) {

method <- training\_params$method }

else {

method <- 'Cosine'}

# Number of nearest neighbors

if (!is.null(training\_params$nn)) {

nn <- as.numeric(training\_params$nn) }

else {

nn <- 10 }

# Number of users to train

if (!is.null(training\_params$n\_users)) {

n\_users <- as.numeric(training\_params$n\_users) }

else {

n\_users <- 190 }

# Read the user book ratings

training\_files = list.files(path=training\_path, full.names=TRUE, pattern='\*.csv')

training\_test\_data = do.call(rbind, lapply(training\_files, read.csv))

# Create book ratings matrix

ratings\_mat = dcast(training\_test\_data, user\_ind~book\_ind, value.var = "BookRating", fun.aggregate=mean)

# Remove user\_ind column

ratings\_mat = as.matrix(ratings\_mat[,-1])

# Reduce the size of the matrix (create a dense matrix)

ratings\_mat = as(ratings\_mat, "realRatingMatrix")

print(paste("Ratings Matrix size: ", nrow(ratings\_mat)))

# Train the model on book ratings - User Based Collaborative Filtering

# For each of the users, identify 10 similar users based on distance between their vectors (defined by book ratings)

rec\_model = Recommender(ratings\_mat[1:n\_users], method = "UBCF", param=list(method=method, nn=nn))

# Generate outputs

#attributes(rec\_model)$class <- 'cosinesimilarity'

save(rec\_model, file=paste(model\_path, 'rec\_model.RData', sep='/'))

print(summary(rec\_model))

write('success', file=paste(output\_path, 'success', sep='/'))}

The following R function is run when SageMaker sends “serve” command (docker run <image> serve) at the time of inference.

# Define scoring function

serve <- function() {

app <- plumb(paste(prefix, 'plumber.R', sep='/'))

app$run(host='0.0.0.0', port=8080)}

Excerpt from plumber.R – take a note at function decorators -- @param req @post /invocations – that turn the function into REST endpoint. The function loads the trained model, creates user book rating matrix, and then predicts top 5 recommendations for user #192

#' Parse input to create user-book matrix and return predictions from model

#' @param req Http request sent

#' @post /invocations

function(req) {

# Specify location of the trained model on the container

prefix <- '/opt/ml'

model\_path <- paste(prefix, 'model', sep='/')

# Load the model

load(paste(model\_path, 'rec\_model.RData', sep='/'), verbose = TRUE)

# Read in index of the user for whom we are predicting recommendations

conn <- textConnection(gsub('\\\\n', '\n', req$postBody))

data <- read.csv(conn)

#print("This is data:", data)

close(conn)

# prepare ratings matrix

ratings\_mat = dcast(data, user\_ind~book\_ind, value.var = "BookRating", fun.aggregate=mean)

# Remove user\_ind column

ratings\_mat = as.matrix(ratings\_mat[,-1])

# Reduce the size of the matrix (create a dense matrix)

ratings\_mat = as(ratings\_mat, "realRatingMatrix")

# Get top 5 recommendations for a user or list of users

pred\_bkratings <- predict(rec\_model, ratings\_mat[192], n=5)

# Return prediction

return(as(pred\_bkratings, "list"))}

The following Dockerfile defines the specifications for training and hosting R model:

FROM ubuntu:16.04

# Install the latest version of R

RUN echo "deb http://cloud.r-project.org/bin/linux/ubuntu xenial/" >> /etc/apt/sources.list

#Add a key to the system so that apt can perform signature checking of the Release File for the added repository to verify its authenticity. The CRAN repository for Ubuntu is signed with the key of “Michael Rutter marutter@gmail.com” with key ID E084DAB9.

RUN apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9

RUN apt-get -y update && apt-get install -y --no-install-recommends \

wget \

r-base \

r-base-dev \

ca-certificates

RUN R -e "install.packages(c('reshape2', 'recommenderlab', 'plumber'), quiet = TRUE)"

COPY Recommender.R /opt/ml/Recommender.R

COPY plumber.R /opt/ml/plumber.R

ENTRYPOINT ["/usr/bin/Rscript", "/opt/ml/Recommender.R", "--no-save"]

Please look at the accompanying SageMaker notebook to review the details on data preparation, training, hosting and inference.

In this chapter, you’ve learned how to process big data to create analytics ready dataset. You’ve also seen how SageMaker automates most of the steps of Machine Learning life cycle, enabling you to build, train, and deploy models seamlessly. Additionally, we’ve illustrated some of the productivity features, such as hyperparameter optimization and experimentation service, that enable data scientists to run multiple experiments and deploy the winning model. Finally, we have looked at how we can bring our own models based on open source machine learning libraries, in addition to bringing your own containers (docker) to SageMaker ecosystem and still leverage all the capabilities of the platform.

References

Improving Recommendation Lists Through Topic Diversification,

Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, Georg Lausen; Proceedings of the 14th International World Wide Web Conference (WWW '05), May 10-14, 2005, Chiba, Japan. To appear. <http://www2.informatik.uni-freiburg.de/~cziegler/BX/>

<https://github.com/awslabs/amazon-sagemaker-examples>

<https://aws.amazon.com/blogs/machine-learning/introduction-to-amazon-sagemaker-object2vec/>

<https://aws.amazon.com/blogs/machine-learning/amazon-sagemaker-now-comes-with-new-capabilities-for-accelerating-machine-learning-experimentation/>

<https://aws.amazon.com/blogs/machine-learning/train-and-host-scikit-learn-models-in-amazon-sagemaker-by-building-a-scikit-docker-container/>