BELGACEM Mohammad-Amine
ZHANG Guantong
CSE 158

A Predictive Model on the Watch Time of Live-Streams

<u>Tasks:</u>

1. Introduction to the dataset and an exploratory analysis

In this report, we will use a dataset of users consuming streaming content on Twitch. The original dataset contains the 3,051,733 broadcast segments of 100,000 users, and each row of the csv file contains the user's ID, the stream's ID, the steamer's username, the time when the stream started, and the time when it ended. The times were recorded every ten minutes, representing a ten-minute period.

```
1,33842865744,mithrain,154,156
1,33846768288,alptv,166,169
1,33886469056,mithrain,587,588
1,33887624992,wtcn,589,591
1,33890145056,jrokezftw,591,594
1,33903958784,berkriptepe,734,737
1,33929318864,kendinemuzisyen,1021,1036
1,33942837056,wtcn,1165,1167
1,33955351648,kendinemuzisyen,1295,1297
1,34060922080,mithrain,2458,2459
1,34062621584,unlostv,2454,2456
1,34077379792,mithrain,2601,2603
1,34078096176,zeon,2603,2604
1,34079135968,elraenn,2600,2601
1,34082259232,zeon,2604,2605
```

Figure 1: CSV File

As an exploratory analysis, we worked on the period of time that users spent on streams. Specifically, we plotted a graph to illustrate the number of users who have been watching a stream for 10, 20, …, 970 minutes.

```
df['interval'] = df['stop'] - df['start']
```

```
interval_count = df.groupby('interval').count()['user']
interval_count
```

```
interval
1      1559891
2       498252
3       263958
4       168944
5       115769
         ...
84           3
88           2
90           2
92           3
97           1
Name: user, Length: 86, dtype: int64
```

```
X = (np.array(list(interval_count.keys()))).reshape(-1, 1)
y = np.array(list(dict(interval_count).values()))
reg = linear_model.LinearRegression().fit(X, np.log(y))
```

Figure 2: Counting and linear regression

Since the distribution is highly skewed and approximately follows the power law, we used the natural logarithm of number of users while doing the linear regression. The regression equation is $y = e^{-0.1376x + 11.6254}$, whose coefficient of determination is 0.9653.

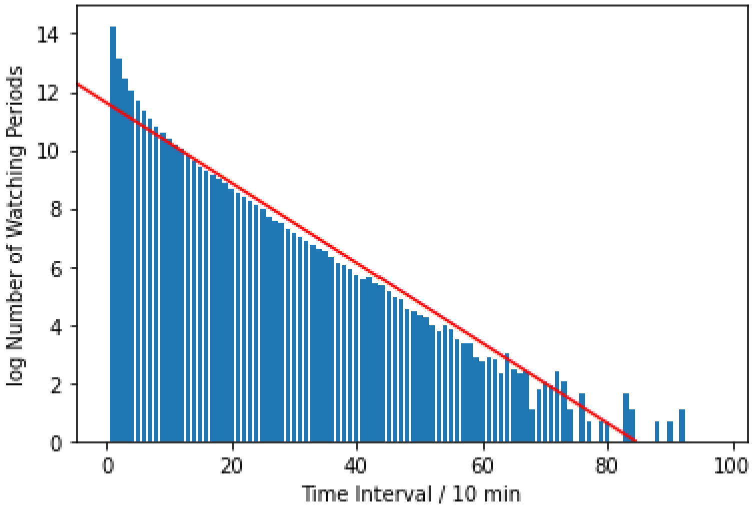

Figure 3: Distribution of numbers of watching periods with different lengths

From this exploratory analysis, we are motivated to start our research on predicting the time interval of broadcast segments by using the user's ID, the stream ID, and the streamer's username.

2. A predictive task on the dataset

As a predictive task, we chose to predict how many minutes a user will spend on a specific stream.

So our dataset looks like this:

```
df.head()
```

	0	1	2	3	4
0	1	33842865744	mithrain	154	156
1	1	33846768288	alptv	166	169
2	1	33886469056	mithrain	587	588
3	1	33887624992	wtcn	589	591
4	1	33890145056	jrokezftw	591	594

Figure 4: Overview of the dataset

To make a good prediction model, we had to create data corresponding to the subtraction of the Start_time and Stop_time columns. This pre-processing of the data will give us a piece of good information serving as a value to predict. This value corresponds to the time spent by me on the stream.

We created this feature this way:

```
df['interval'] = df[4] - df[3]
```

```
df.head()
```

	0	1	2	3	4	interval
0	1	33842865744	mithrain	154	156	2
1	1	33846768288	alptv	166	169	3
2	1	33886469056	mithrain	587	588	1
3	1	33887624992	wtcn	589	591	2
4	1	33890145056	jrokezftw	591	594	3

Figure: Creation of the "interval" column.

We decided to shuffle our database to avoid overfitting because the database is in the user id order. And when we split our database to evaluate it, we won't have user id values that are different and large enough for the evaluation.
Then, we keep 100,000 random lines of the dataset for our future prediction task.

```
df_sample = df.sample(100000).reset_index(drop=True)
df_sample.columns = ['user', 'stream_id', 'streamer', 'start', 'stop', 'interval']
df_sample
```

	user	stream_id	streamer	start	stop	interval
0	15080	34046829072	aghhi	2254	2255	1
1	63220	33858981296	renzo	298	302	4
2	95643	34027272624	shroud	2049	2050	1
3	64676	34202486704	niccasso	3905	3910	5
4	26609	34048312656	vechiron	2280	2282	2
...	...	...	...	...	...	...
99995	6118	33925134976	soleil5813	1012	1013	1
99996	52630	34044085264	wtf_winds123	2220	2229	9
99997	60196	34187805376	valechoberk	3743	3748	5
99998	48421	33867116512	jrokezftw	401	404	3
99999	93542	34043605328	gamerbee	2236	2237	1

100000 rows × 6 columns

Figure 5: New database

And we save our new database:

```
df_sample.to_csv('100k_a_sample.csv', index=False)
```

3. Select and design an appropriate model

We start to load our database:

```
data = []
with open('100k_a_sample.csv', mode ='r') as file:
    file.readline()
    csvFile = csv.reader(file)
    for lines in csvFile:
        data.append(lines[0:3] + [float(lines[-1])])
```

First, to evaluate our data, we will divide our data into a training set of the first 80,000 rows, a valid test of 10 000 rows and a test set of the last 10,000. We will then predict with the trained model on the training data set on the test set.

```
train = data[:80000]
valid = data[80000:90000]
test = data[90000:]
train_df = df_sample[:80000]
valid_df = df_sample[80000:90000]
test_df = df_sample[90000:]
```

We start by creating different dictionaries containing the average viewing times per stream_id, per streamer, and per user.

```
from collections import defaultdict

ratings_per_user = defaultdict(list)
for d in train:
    ratings_per_user[d[0]].append(d[-1])
ratings_per_user = {u: np.mean(ratings_per_user[u]) for u in ratings_per_user}

ratings_per_stream = defaultdict(list)
for d in train:
    ratings_per_stream[d[1]].append(d[-1])
ratings_per_stream = {u: np.mean(ratings_per_stream[u]) for u in ratings_per_stream}

ratings_per_streamer = defaultdict(list)
for d in train:
    ratings_per_streamer[d[2]].append(d[-1])
ratings_per_streamer = {u: np.mean(ratings_per_streamer[u]) for u in ratings_per_streamer}
```

We will create our model using the values of user_id, and stream_id for the features and the value of the difference between Start_time and Stop_time for our label.

If the value is present in the dictionary, then we will retrieve the average value in the corresponding dictionary. Otherwise, we return the global average value.

```
def feature(datum):
    global_mean = train_df['interval'].mean()

    if datum[0] in ratings_per_user:
        user_avg = ratings_per_user[datum[0]]
    else:
        user_avg = global_mean

    if datum[1] in ratings_per_stream:
        stream_avg = ratings_per_stream[datum[1]]
    else:
        stream_avg = global_mean

    if datum[2] in ratings_per_streamer:
        streamer_avg = ratings_per_streamer[datum[2]]
    else:
        streamer_avg = global_mean

    feat = [1, user_avg, stream_avg, streamer_avg]

    return feat
```

```
feature(train[1])
```

```
[1, 2.5, 4.0, 4.0]
```

Figure; Creation of our features

Then we create all our X and y variables:

```
X_train = [feature(x) for x in train]
y_train = [x[-1] for x in train]

X_valid = [feature(x) for x in valid]
y_valid = [x[-1] for x in valid]

X_test = [feature(x) for x in test]
y_test = [x[-1] for x in test]
```

Figure: Creation of train sets, validation sets, and test sets

We start with a simple model, the Linear regression model. We are going to evaluate it with mean standard error.

```
model_ = linear_model.LinearRegression()
model_.fit(X_train,y_train)
```

We then calculate the MSE:

```
def MSE(predictions, labels):
    differences = [(x-y)**2 for x,y in zip(predictions,labels)]
    return sum(differences) / len(differences)
```

```
pred_valid = model_.predict(X_valid)
MSE(y_valid, pred_valid)
```

```
20.01017935631818
```

We see that our model has an MSE of approximately 20 (it depends on how the dataset is sampled at the beginning). The linear regression model is quite simple because it will only look for a relationship between our X's, which are our features, and our y which is our predictor.

We then decided to implement a latent factor model with TensorFlow.

So our data looks like this:

```
Entrée [190]:  data
```

```
Out[190]: [['24152', '33941974112', 'ladyduckfan', 1.0],
           ['14017', '34211667200', 'zeon', 1.0],
           ['62588', '34374431808', 'esl_dota2', 6.0],
           ['23140', '34017003216', 'maxalibur', 1.0],
           ['33584', '33918826784', 'deucenickalo', 7.0],
           ['32195', '34121079568', 'educatedeartw', 2.0],
           ['88044', '34414130864', 'lnz3', 2.0],
           ['94012', '33842536576', 'crystal_lol', 1.0],
           ['29279', '33836408736', 'mandiocaa1', 1.0],
           ['54303', '34380812000', 'tcasupreme', 2.0],
           ['89125', '34062761376', 'locklear', 1.0],
           ['59482', '34305339616', 'viperprohs', 1.0],
           ['87102', '34020681568', 'crazymann', 3.0],
           ['98892', '34352587728', 'auraina', 2.0],
           ['32679', '34373416272', 'gabrielcro', 1.0],
           ['24528', '33872516560', 'kung', 8.0],
           ['20493', '34104770528', 'drdisrespect', 2.0],
           ['67626', '33993051536', 'thereisnofuture', 1.0],
           ['73629', '34009693632', 'magenta62', 11.0],
```

We start by creating everything we need:

```
userIDs = {}
itemIDs = {}
interactions = []

for d in data:
    u = d[0]
    i = d[1]
    r = d[3]
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)
    interactions.append((u,i,r))
```

```
len(interactions)
```

```
100000
```

We then divide our data into a train set and a test set.

```
interactionsTrain  = interactions[:90000]
interactionsTest  = interactions[90000:]
```

We then start the implementation of the model.

```
mu = sum([r for _,_,r in interactionsTrain]) / len(interactionsTrain)
```

```
optimizer = tf.keras.optimizers.Adam(0.01)
```

mu is the average value to be used by default.
We also instantiate our gradient descent optimizer, Adam.

Then we create our Latent factor model class with TensorFlow.

```
class LatentFactorModelBiasOnly(tf.keras.Model):
    def __init__(self, mu, lamb):
        super(LatentFactorModelBiasOnly, self).__init__()
        # Initialize to average
        self.alpha = tf.Variable(mu)
        # Initialize to small random values
        self.betaU = tf.Variable(tf.random.normal([len(userIDs)],stddev=0.001))
        self.betaI = tf.Variable(tf.random.normal([len(itemIDs)],stddev=0.001))
        self.lamb = lamb

    # Prediction for a single instance (useful for evaluation)
    def predict(self, u, i):
        p = self.alpha + self.betaU[u] + self.betaI[i]
        return p

    # Regularizer
    def reg(self):
        return self.lamb * (tf.reduce_sum(self.betaU**2) +\
                            tf.reduce_sum(self.betaI**2))

    # Prediction for a sample of instances
    def predictSample(self, sampleU, sampleI):
        u = tf.convert_to_tensor(sampleU, dtype=tf.int32)
        i = tf.convert_to_tensor(sampleI, dtype=tf.int32)
        beta_u = tf.nn.embedding_lookup(self.betaU, u)
        beta_i = tf.nn.embedding_lookup(self.betaI, i)
        pred = self.alpha + beta_u + beta_i
        return pred

    # Loss
    def call(self, sampleU, sampleI, sampleR):
        pred = self.predictSample(sampleU, sampleI)
        r = tf.convert_to_tensor(sampleR, dtype=tf.float32)
        return tf.nn.l2_loss(pred - r) / len(sampleR)
```

Then, we create our train function:

BELGACEM Mohammad-Amine
ZHANG Guantong
CSE 158

```
def trainingStepBiasOnly(model, interactions):
    Nsamples = 50000
    with tf.GradientTape() as tape:
        sampleU, sampleI, sampleR = [], [], []
        for _ in range(Nsamples):
            u,i,r = random.choice(interactions)
            sampleU.append(userIDs[u])
            sampleI.append(itemIDs[i])
            sampleR.append(r)

        loss = model(sampleU,sampleI,sampleR)
        loss += model.reg()
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients((grad, var) for
        (grad, var) in zip(gradients, model.trainable_variables)
        if grad is not None)
    return loss.numpy()
```

We start training our model:

```
modelBiasOnly = LatentFactorModelBiasOnly(mu,0.00001)
```

```
for i in range(50):
    obj = trainingStepBiasOnly(modelBiasOnly, interactionsTrain)
    if (i % 10 == 9): print("iteration " + str(i+1) + ", objective = " + str(obj))
```

```
iteration 10, objective = 8.384841
iteration 20, objective = 8.236091
iteration 30, objective = 8.508019
iteration 40, objective = 8.189702
iteration 50, objective = 8.056442
```

Then, we get all the predictions and labels to evaluate our model.

```
pred = [modelBiasOnly.predict(userIDs[d[0]], itemIDs[d[1]]).numpy() for d in interactionsTest]
pred[0:5]
```

```
[3.3983119, 3.6085575, 3.1761374, 3.0208411, 3.006167]
```

```
label = [d[2] for d in interactionsTest]
label[0:5]
```

```
[9.0, 3.0, 1.0, 2.0, 1.0]
```

Finally, we calculated our MSE and we saw that we had improved our performance by minimizing our MSE.

```
MSE(pred,label)
```

```
17.805034688583753
```

We adapted the learning rate of the model and tested different batch sizes. We came to the conclusion that a batch size of 50 and a learning rate of 0.01 was the most efficient.

Before keeping the latent factor model, we tested a logic regression model but it was not adapted and we had a rather bad performance (our MSE was notably high).

The first weak point we can state about linear regression is that it is a model that has difficulty fitting complicated data. But its strong point is that it is easily understandable and explainable and you can easily avoid overfitting with this model.

The Latent Factor Model will allow us to improve our performances because it will reduce the dimension for the real-valued prediction. It is more accurate for new pairs of users and items.

4. Related literature, similar datasets, and comparisons

Our dataset comes from *Recommendation on Live-Streaming Platforms: Dynamic Availability and Repeat Consumption* [1]. In this article, Rappaz et al. pointed out that a unique property of the live-streaming data - not all items are available to a user since many streams are concurrent. In this setting, they used a matrix to represent the availability of items at the interaction time by using the Time Start and Time Stop in the dataset. The state-of-the-art methods currently employed on this type of data is to rank the items in each step in the sequence embedding, and to select available items using a self-attention mechanism [1].

With the streaming data, we are able to estimate the audience's preference on a certain live stream by predicting the period of time they will spend on the stream. The difference between our work and the research done by Rappaz et al. is that we built a predictive model only based on individual time intervals without considering the interactions between the training set. As a result, the predictions made by our modal may contain impossible results. For example, a user was watching stream S1 from time 20 to 25 according to the training data, while we still predict that the same user will watch stream S2 which starts at time 23 for 7 units of time (until time 30).

Similar datasets can be found on TwitchTracker [2]. Instead of recording each user's watch times, the website shows the aggregate watch time of all users over time, illustrating the trend of watching.

5. Results and conclusions

Regarding the results obtained, our LFM model has a smaller MSE than the basic linear regression, so the model performs better.
We also conclude that the most important features are the user_id and stream_id. We realized that the streamer name was not used and did not influence the result.
Our MSE is basically high due to the variance of our values but compared to our different MSE values depending on the model, we finally performed well and managed to optimize our model well.

As we said, the latent factor model was the most adapted for our dataset because the biggest problem we encountered was that it was possible to have in the test set user/item pairs never met in the train set. And on this point, the linear regression model had poor performance while the latent factor model is a more adaptive model and comfortable with never met user/item pairs.

References:

[1] Recommendation on Live-Streaming Platforms: Dynamic Availability and Repeat Consumption, Jérémie Rappaz, Julian McAuley and Karl Aberer, RecSys, 2021

[2] https://twitchtracker.com/statistics/watch-time