Summary of Model results from HW1 and 2 at the end of the notebook.

In [ ]:
```
import numpy as np
import pandas as pd
import plotly.express as px
```

In [ ]:
```
# Use filtered train and test from HW1 used for the BL models.
test = pd.read_csv("filtered_test.csv")
train = pd.read_csv("filtered_train.csv")
```

Rating bias, for a user and item is defined as

$$b_{ui} = \mu + b_i + b_u$$

where:

- μ = overall average
- b_i, b_u = observed deviations for item i and user u from the average.

The predicted rating, with bias, is

$$\hat{r}_{ui} = \mu + b_i + b_u + q_i^T p_u = b_{ui} + q_i^T p_u$$

The updated objective is:

$$\min_{q^*,p^*} \sum_{(u,i)\in\kappa} (r_{ui} - b_{ui} - q_i^T p_u)^2 + \lambda \left(\|q_i\|^2 + \|p_u\|^2 + b_u^2 + b_i^2 \right)$$

I will use 10 latent factors

In [ ]:
```
mean_r = train['rating'].mean()
user_mean_dev = train.groupby('user')['rating'].mean() - mean_r
item_mean_dev = train.groupby('movie')['rating'].mean() - mean_r
user_ids = pd.concat([test['user'], train['user']]).unique()
```

In [ ]:
```
user_mean_dev
```

Out[ ]:
```
user
10         -0.113601
188        -0.100212
195         0.183011
307         0.080814
462        -0.439013
               ...
2649034    -0.944306
2649093    -0.623805
2649116    -0.003397
2649370     0.533590
2649426     0.516603
Name: rating, Length: 39476, dtype: float64
```

In [ ]:
```
gamma = 0.02
lambda_ = 0.04
```

In [ ]:
```
p = np.ones((len(user_ids), 10)) - .9
q = np.ones((17770, 10)) - .9
```

In [ ]:
```
user_index = {user: i for i, user in enumerate(user_ids)}
movie_index = {movie: i for i, movie in enumerate(train['movie'].unique())}
```

In [ ]:
```
import time
start = time.time()

for i in range(len(train)):
    user = train.iloc[i]['user']
    movie = train.iloc[i]['movie']
    r_ui = train.iloc[i]['rating']
    q_row = q[movie_index[movie], :]
    p_row = p[user_index[user], :]

    r_ui_hat = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q_row, p_row)

    p[user_index[user], :] += gamma * ((r_ui - r_ui_hat) * q_row - lambda_ * p_row)

    q[movie_index[movie], :] += gamma * ((r_ui - r_ui_hat) * p_row - lambda_ * q_row)

    if i % 1e6 == 0:
        print(i)

end = time.time()
print(end - start)
```

```
0
1000000
2000000
3000000
571.3567724227905
```

In [ ]:
```
np.save('p_matrix_bias_A1_filtered.npy', p)
np.save('q_matrix_bias_A1_filtered.npy', q)
```

In [ ]:
```
p
```

Out[ ]:
```
array([[0.10446609, 0.10446609, 0.10446609, ..., 0.10446609, 0.10446609,
        0.10446609],
       [0.09303116, 0.09303116, 0.09303116, ..., 0.09303116, 0.09303116,
        0.09303116],
       [0.10620898, 0.10620898, 0.10620898, ..., 0.10620898, 0.10620898,
        0.10620898],
       ...,
       [0.10055095, 0.10055095, 0.10055095, ..., 0.10055095, 0.10055095,
        0.10055095],
       [0.09966185, 0.09966185, 0.09966185, ..., 0.09966185, 0.09966185,
        0.09966185],
       [0.09953351, 0.09953351, 0.09953351, ..., 0.09953351, 0.09953351,
        0.09953351]])
```

Implicit Feedback

With implicit feedback, the new predicted rating is

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + |I_u|^{-\frac{1}{2}} \sum_{j \in I_u} y_j \right)$$

where:

- I_u = the set of all items rated by user u

- y_j = Our new set of user factors that capture implicit ratings.

"The dataset does not only tell us the rating values, but also which movies users rate, regardless of how they rated these movies. In other words, a user implicitly tells us about her preferences by choosing to voice her opinion and vote a (high or low) rating. This reduces the ratings matrix into a binary matrix, where "1" stands for "rated", and "0" for "not rated". Admittedly, this binary data is not as vast and independent as other sources of implicit feedback could be. Nonetheless, we have found that incorporating this kind of implicit data – which inherently exist in every rating based recommender system – significantly improves prediction accuracy. Some prior techniques, such as Conditional RBMs [18], also capitalized on the same binary view of the data."

For implicit feedback, I will use the normalized sum of q vectors from the bias only model for the set of movies rated by a user. That is, each of the 10 item factors gets combined across the set of movies each user rated, forming the implicit feedback matrix.

In [ ]:
```
p_implicit = np.zeros((len(user_ids), 10))
w = 0
for i in set(train['user']):
    user_df = train[train['user'] == i]
    movie_indices = [movie_index[movie] for movie in user_df['movie']]
    p_implicit[user_index[i], :] = user_df.shape[0]**(-0.5) * np.sum(q[movie_indices, :], axis=0)
    w+=1
    if w % 10000 == 0:
        print(w)
```

```
10000
20000
30000
```

In [ ]:
```
np.save('p_implicit_A2_filtered.npy', p_implicit)
```

In [ ]:
```
p_implicit = np.load('p_implicit_A2_filtered.npy')
```

In [ ]:
```
p = np.ones((len(user_ids), 10)) - .9
q = np.ones((17770, 10)) - .9
```

In [ ]:
```
gamma = 0.02
```

Out[ ]:
```
array([[0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1],
       ...,
       [0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1],
       [0.1, 0.1, 0.1, ..., 0.1, 0.1, 0.1]])
```

In [ ]:
```
# stop overflow error
clip = 1
```

In [ ]:
```
for i in range(len(train)):
    user = train.iloc[i]['user']
    movie = train.iloc[i]['movie']

    r_ui_hat = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q[movie_index[mo

    p[user_index[user], :] += np.clip(gamma * ((train.iloc[i]['rating'] - r_ui_hat) * q[movie_index[

    q[movie_index[movie], :] += np.clip(gamma * ((train.iloc[i]['rating'] - r_ui_hat) * p[user_index
```

```
    if i % 1e6 == 0:
        print(i)
```

```
0
1000000
2000000
3000000
```

In [ ]:
```
np.save('p_A2_filtered.npy', p)
np.save('q_A2_filtered.npy', q)
```

Time-Varying Baseline Predictors

$$b_i(t) = b_i + b_{i,Bin(t)}$$
$$dev_u(t) = sign(t - t_u) \cdot |t - t_u|^\beta$$
$$b_u^{(1)}(t) = b_u + \alpha_u \cdot dev_u(t)$$
$$b_u^{(2)}(t) = b_u + \frac{\sum_{l=1}^{k_u} \exp(-\gamma|t - t_l^u|) b_{t_l}^u}{\sum_{l=1}^{k_u} \exp(-\gamma|t - t_l^u|)}$$

Where bins in Collaborative Filtering with Temporal Dynamics were acceptable when they contained about 10 weeks (~30 bins in the entire dataset) and $\beta = 0.4$ was determined via CV. α_u can similarly be learned from the data, but I will set it to 0.025, which limits the maximum absolute adjustment to the baseline user rating to be about 0.4, given the approximate largest date range a user can make ratings in this dataset (i.e., 10 weeks * 7 days * 15 bins on either side = approximately 16 * 0.025 = 0.4)

$t_u = \text{mean rating date for user}$ (days)

$b_u^{(1)}$ provides a simpler, less flexible approach to time changing user bias than $b_u^{(2)}$ piecewise approach. I will use $b_u^{(1)}$ for simplicity.

There are additional ways of modelling "concept drift" in the baseline predictors. For instance, one could add b_{u,t} to b^{(1)} to capture average ratings on days a participant rated multiple movies at once, since those apparently hover around singular values. This apparently increases accuracy. However, due to the increased complexity of this (adding on average 40 parameters per user), I will ignore it for the purposes of this homework.

$$b_{ui}(t) = \mu + b_i(t) + b_u^{(1)}(t)$$

where:

- μ = overall average
- b_i, b_u = observed deviations for item i and user u from the average.

The predicted rating, with bias, is

$$\hat{r}_{ui} = b_{ui}(t) + q_i^T p_u$$

For simplicity, I will bin my item ratings by quarter

In [ ]:
```
train['datetime'] = pd.to_datetime(train['date'])
train['year'] = train['datetime'].dt.year
train['quarter'] = train['datetime'].dt.to_period('Q')
```

```
min_date = train['datetime'].min()

train['day'] = (train['datetime'] - min_date).dt.days + 1
```

In [ ]:
```
user_mean_day = train.groupby('user')['day'].mean()
```

In [ ]:
```
item_mean_per_quarter = train.groupby(['movie', 'quarter'])['rating'].mean()
item_mean_dev_by_quarter = item_mean_per_quarter - item_mean_dev - mean_r
item_mean_dev_by_quarter
```

Out[ ]:
```
movie  quarter
8      2004Q3     0.694907
       2004Q4     0.728241
       2005Q1     0.023348
       2005Q2     0.017083
       2005Q3    -0.110093
                    ...
17769  2004Q3    -0.048701
       2004Q4    -0.064291
       2005Q1    -0.028997
       2005Q2     0.190629
       2005Q3    -0.011755
Name: rating, Length: 21348, dtype: float64
```

In [ ]:
```
alpha = 0.025
beta = 0.4
```

In [ ]:
```
p = np.ones((len(user_ids), 10)) - .9
q = np.ones((17770, 10)) - .9
p_fac = np.ones((len(user_ids), 10)) - .9
q_fac = np.ones((17770, 10)) - .9
```

In [ ]:
```
for i in range(len(train)):
    user = train.iloc[i]['user']
    movie = train.iloc[i]['movie']
    r_ui = train.iloc[i]['rating']
    quarter = train.iloc[i]['quarter']
    day = train.iloc[i]['day']
    q_row = q[movie_index[movie], :]
    p_row = p[user_index[user], :]
    q_row_fac = q_fac[movie_index[movie], :]
    p_row_fac = p_fac[user_index[user], :]

    dev_ut = np.sign(day - user_mean_day.loc[user]) * (abs(day - user_mean_day.loc[user])**beta)
    b_ut = user_mean_dev.loc[user] + alpha * dev_ut

    r_ui_hat = mean_r + b_ut + item_mean_dev_by_quarter.loc[(movie, quarter)] + item_mean_dev.loc[mo
    p[user_index[user], :] += np.clip(gamma * ((r_ui - r_ui_hat) * q_row - lambda_ * p_row), -clip,
    q[movie_index[movie], :] += np.clip(gamma * ((r_ui - r_ui_hat) * p_row - lambda_ * q_row), -clip

    r_ui_hat_fac = mean_r + b_ut + item_mean_dev_by_quarter.loc[(movie, quarter)] + item_mean_dev.lo
    p_fac[user_index[user], :] += np.clip(gamma * ((r_ui - r_ui_hat_fac) * q_row_fac - lambda_ * p_r
    q_fac[movie_index[movie], :] += np.clip(gamma * ((r_ui - r_ui_hat_fac) * p_row_fac - lambda_ * q

    if i % 1e6 == 0:
        print(i)
```

```
0
1000000
2000000
3000000
```

In [ ]:
```
np.save('p_A3_filtered_time_bias.npy', p)
np.save('q_A3_filtered_time_bias.npy', q)
```

In [ ]:
```
np.save('q_A4_filtered_time.npy', q_fac)
np.save('p_A4_filtered_time.npy', p_fac)
```

Time-Varying Baseline Predictors and Factors

The above also added time varying user factors as well in separate p and q matrices (_fac)

$$p_{uk}(t) = p_{uk} + \alpha_{uk} \cdot dev_u(t)$$

Where k represents a latent factor. We can similarly add a $p_{uk,t}$ term, as we could have to the user bias term. However, this once again adds much more complexity in terms of number of parameters, which I will ignore for the purposes of this homework.

Test Set Performance

Time information is ignored for test set predictions. The function the time-based parameters served was to "absorb" transient fluctuations in the training data that do not reflect more stable representations of true rating patterns

In [ ]:
```
test = pd.read_csv("filtered_test.csv")
print(len(test['user'].unique()));print(len(test['movie'].unique()))
```

```
925
100
```

In [ ]:
```
test = test[test['movie'].isin(train['movie'])]
print(len(test['user'].unique()));print(len(test['movie'].unique()))
```

```
925
100
```

In [ ]:
```
test = test[test['user'].isin(train['user'])]
print(len(test['user'].unique()));print(len(test['movie'].unique()))
```

```
922
100
```

In [ ]:
```
p_bias = np.load('p_matrix_bias_A1_filtered.npy')
q_bias = np.load('q_matrix_bias_A1_filtered.npy')

p_ifb = np.load('p_implicit_A2_filtered.npy') # implicit feedback
p_implicit = np.load('p_A2_filtered.npy')
q_implicit = np.load('q_A2_filtered.npy')

p_time_bias = np.load('p_A3_filtered_time_bias.npy')
q_time_bias = np.load('q_A3_filtered_time_bias.npy')

p_time_factor = np.load('p_A4_filtered_time.npy')
q_time_factor = np.load('q_A4_filtered_time.npy')

r_ui = test['rating'].values
r_ui_hat_bias = np.empty(len(test))
r_ui_hat_implicit = np.empty(len(test))
r_ui_hat_time_bias = np.empty(len(test))
r_ui_hat_time_factor = np.empty(len(test))
```

In [ ]:
```
def calculate_rmse(rui, rui_hat):
    return np.sqrt(np.mean((rui - rui_hat) ** 2))

def calculate_mape(rui, rui_hat):
    return np.mean(np.abs((rui - rui_hat) / rui))
```

In [ ]:
```
for i in range(len(test)):
    user = test.iloc[i]['user']
    movie = test.iloc[i]['movie']
    q_row_bias = q_bias[movie_index[movie], :]
    p_row_bias = p_bias[user_index[user], :]
    q_row_implicit = q_implicit[movie_index[movie], :]
    p_row_implicit = p_implicit[user_index[user], :]
    p_row_ifb = p_ifb[user_index[user], :]
    q_row_time_bias = q_time_bias[movie_index[movie], :]
    p_row_time_bias = p_time_bias[user_index[user], :]
    q_row_time_factor = q_time_factor[movie_index[movie], :]
    p_row_time_factor = p_time_factor[user_index[user], :]

    r_ui_hat_bias[i] = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q_row_bi
    r_ui_hat_implicit[i] = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q_ro
    r_ui_hat_time_bias[i] = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q_r
    r_ui_hat_time_factor[i] = mean_r + user_mean_dev.loc[user] + item_mean_dev.loc[movie] + np.dot(q

    if i % 1e6 == 0:
        print(i)
```

```
0
```

In [ ]:
```
rmse_bias = calculate_rmse(r_ui, r_ui_hat_bias)
mape_bias = calculate_mape(r_ui, r_ui_hat_bias)

rmse_implicit = calculate_rmse(r_ui, r_ui_hat_implicit)
mape_implicit = calculate_mape(r_ui, r_ui_hat_implicit)

rmse_time_bias = calculate_rmse(r_ui, r_ui_hat_time_bias)
mape_time_bias = calculate_mape(r_ui, r_ui_hat_time_bias)

rmse_time_factor = calculate_rmse(r_ui, r_ui_hat_time_factor)
mape_time_factor = calculate_mape(r_ui, r_ui_hat_time_factor)

# Code BL results from HW1 and combine with HW2
results_wide = pd.DataFrame({
    'model': ['ibcf', 'svd', 'xgb', 'bias', 'implicit', 'time_bias', 'time_factor'],
    'rmse': [1.0898492, 0.8421464, 0.9693462, rmse_bias, rmse_implicit, rmse_time_bias, rmse_time_fa
    'mape': [0.2556977, 0.2444427, 0.2869236, mape_bias, mape_implicit, mape_time_bias, mape_time_fa
})

print(results_wide)
```

```
        model      rmse      mape
0        ibcf  1.089849  0.255698
1         svd  0.842146  0.244443
2         xgb  0.969346  0.286924
3        bias  0.960755  0.280269
4    implicit  0.992224  0.288865
5   time_bias  0.957627  0.274555
6  time_factor  0.981942  0.285987
```

In [ ]:
```
results_long = results_wide.melt(id_vars='model', var_name='metric', value_name='value')

fig = px.bar(results_long, x='model', y='value', color='model', facet_col='metric', title='Model Per
fig.show()
```

Summary

From the performance metrics on the test set, the SVD baseline model had the best performance in terms of both RMSE and MAPE. Of the 4 Koren Models, the model with baseline time-dependent biases performed the best in terms of both metrics. The only model this "time_bias" model performed worse than was SVD. XGBoost performed better than both the Koren model with implicit feedback and the Koren model with both the baseline time-dependent biases and the time-dependent factors.

While this is disappointing, it highlights some interesting points about these models.

For SVD, these results suggest that it performs well on a smaller dataset relative to the other models.

For Koren's models, these results suggest numerous highlights about the training process. One point is that I only used one iteration through the training set to update the factor parameters. This means that some parameters may have had much farther to move to reach their optimal values, which I avoided for the sake of time. Indeed, many parameters in the Koren models did not move far away from their parametrization at 0.1, far more itereations (at least 10 or 20 more) could have been used to converge to closer to optimal values. In line with this point, I used SGD for my learning algorithm, when alternating least squares is known to be better performing. Additionally, the full dataset may have identified more generalizable movie and user features vectors. Another point is that all factors get updated together given the update equations, which means the values for each latent factor are the same. This suggests more interesting and useful factors might have emerged if a more random initialization was used rather than all 0.1s, or if the latent factors were all updated separately. Lastly, it is remarkable that the time_factor model performs worse than the time_bias model, considering that the time_bias model is nested within the time_factor model. This suggests model performance would have been improved by learning the value of α from the dataset or using the splines method, as well as the fact that the additional complexity to model can hurt performance, likely due to the smaller subset of data used.

In the full dataset:

- There are about 80M ratings in the training set and 20M in the test set.
- There are 405,041 unique users in the training data set
- There are 349,312 unique users in the testing data set
- There are 17,424 unique movies in the training data set
- There are 17,757 unique movies in the testing data set

In the filtered dataset:

- There were 39476 users and 1699 movies
- There were 925 users (filtered down to 922 users that were also in the filtered train dataset) and 100 movies.