Project Definition

Project Overview

I've always disliked the recommendations that Crunchyroll, a popular anime streaming site, has given me when searching for a new anime show to watch. Most of their recommendations are for new shows that do not have a big following or they recommend all time highest rated shows that are also recommended on every other platform. I wanted to use what I learned about creating recommendation engines to build my own. This project will be a stepping stone into building (in the future) my own web application that runs off of a FunkSVD model (a machine learning algorithm).

Project Statement

Every recommendation engine has three types of recommendations. The first is knowledge based. It's like the filter buttons on the top of the screen at any online marketplace that lets a user sort items by price, date, material, etc. The types of filters that I used for this project are based on genre (a show could be a part of multiple genres) and decade (which decade did the show *first* aired).

The second type of recommendations come from the content itself, called content based recommendations. If a user rates newer romantic shows higher than older military based shows, then newer romantic shows will be recommended.

The third type of recommendations come from other similar users, called collaborative based recommendations. This is where machine learning ties into the project. With an algorithm called FunkSVD (Funk Singular Value Decomposition), it is possible to predict what a user might rate a show based on what other similar users rated it. If the algorithm predicts that a user would like a show that they haven't seen before, then that show should be recommended.

Metrics

To evaluate the recommendations given by FunkSVD, mean squared error (MSE) will be used. MSE is straightforward and is best explained through an example.

Here is the equation for reference.

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

	Actual
User id	Jujutsu Kaisen
1	8
2	2
3	6
4	8
5	
6	10
7	3
8	4
9	6
10	5

	Predicted
User id	Jujutsu Kaisen
1	7
2	3
3	8
4	7
5	4
6	9
7	4
8	3
9	5
10	5

Squared Error
Jujutsu Kaisen
1
1
4
1
16
1
1
1
1
0

MSE
2.7

As an example, we take a show called Jujutsu Kaisen. There is a list of users who have given the show an actual rating. MSE takes the actual rating, subtracts the predicted rating and then squares the subtraction to get the squared error for each user. This is the top of the mathematical equation $(Y_{actual} - Y_{pred})^2$. Then the sum of all of the squared errors are divided by the total number of ratings there are to get the mean squared error. *The closer to 0 the MSE is, the more accurate the predictions are.* The goal would be to get MSE to as close to 0 as possible.

Analysis

Data Exploration - Data Visualization

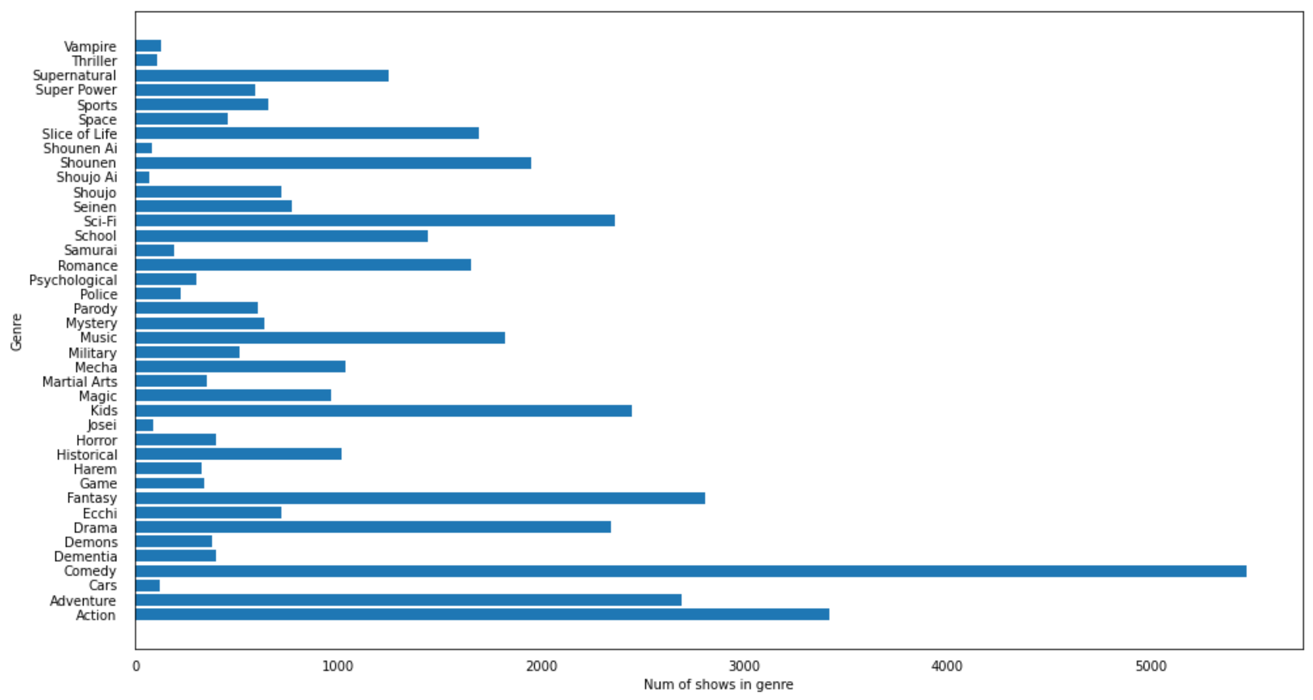
The full datasets I used for this project can be found here:
https://www.kaggle.com/datasets/marlesson/myanimelist-dataset-animes-profiles-reviews

Summary of data:
I chose this dataset because of its usability. The data was scraped from MyAnimeList.net which is a popular website for anime fans to rate and discuss any and all anime shows. All the data needed to create a recommendation engine was present. So working off of these csv files will make data preprocessing a lot easier.

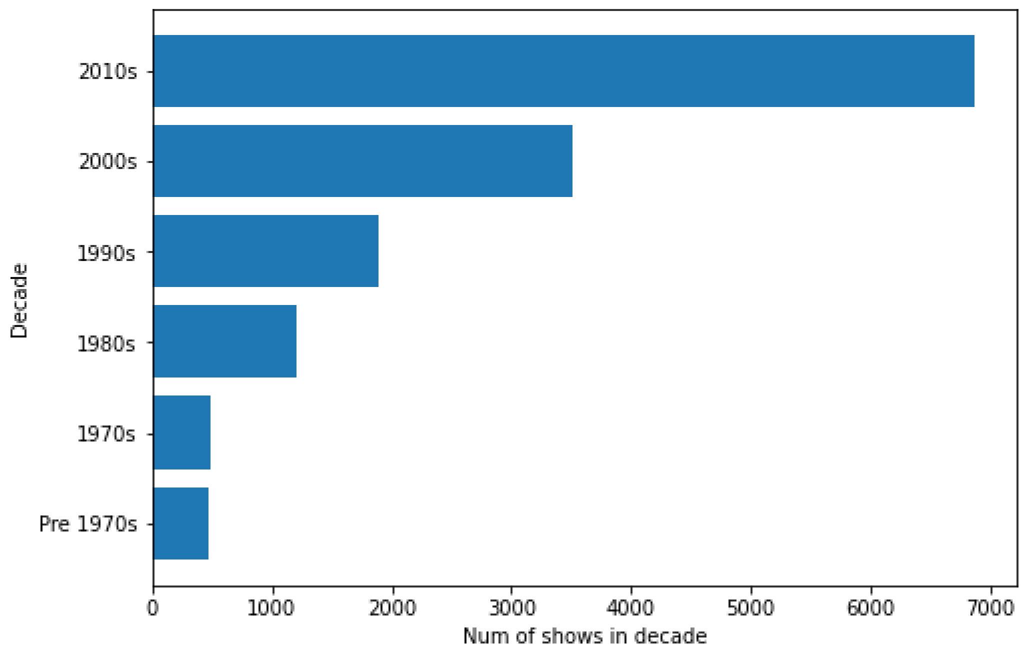
Only two of the three datasets will be needed for this project. For knowledge and content-based recommendations, only the anime.csv file will be needed. The reviews.csv file will be needed for the collaborative recommendations.

In total, there are 130,519 reviews. Yet only 8113 different anime shows are rated. This will make collaborative recommendations a little tricky but no review is missing any data.

Distribution of Genres



Distribution of Shows by Decade



There are about 14,500 different anime shows that span 40 different genres. The top genres being Comedy, Action, Adventure and Fantasy. Many shows will fall into these genres so there should be no shortage of recommendation to be given for them. Each decade, anime has been gaining in popularity which results in more shows being made. About half of the shows being used in this project are from the 2010’s.

Methodology

Data Preprocessing

The data preprocessing can be broken up into two sections. One for the animes.csv file and the other for the reviews.csv file. A summary of the steps will be listed below and full details of all the preprocessing steps can be found in the Jupyter Notebook.

animes.csv

1. Keep the 6 columns that are needed for knowledge/content based recommendations (uid, title, genre, aired, ranked, score). Get rid of the rest.
2. Check for any duplicate rows. If they exist, get rid of them
3. Check for any null values in ‘ranked’ and ‘score’ columns. If they exist, get rid of them.
4. Split the genre column.
    a. Create a separate column for each genre that exists in the data.
    b. For each show in the dataset, put a 1 in the genre columns that the show belongs to, 0 otherwise.
5. Split the aired column.
    a. Get the initial air date (year only).
    b. Assign it to the appropriate decade.
    c. Create a separate column for each decade.
    d. For each show in the dataset, put a 1 in the decade column that the show belongs to, 0 otherwise.

Having the separate columns for each genre and decade makes the implementation of the knowledge/content based recommendations easy and makes the code readable.

reviews.csv

There are only a few preprocessing steps that need to be taken because the goal of this step is to turn this csv file into a user-item matrix.

1. Keep the three columns needed (profile, anime_uid, score). Get rid of the rest.
2. Check for duplicate rows. If they exist, get rid of them.
3. Check for null values in the ‘score’ column. If they exist, get rid of them.

Implementation

The implementation can be followed in great detail in the Jupyter Notebook.

Section 1 - Knowledge Based Recommendations

This is the simplest form of recommending content to users because the user tells us what they want based on the options we provide. There are only 3 'filters' the users can sort by (rank, genre, date). So the steps for the section are:

1. Create get_top_ranked() function
    a. Sorts the data by rank (in ascending order) and returns the number of recommendations the user wants.
2. Create get_top_ranked_genre() function
    a. Filters out all other shows that are not in a given genre and returns the number of recommendations the user wants (sorted by rank).
3. Create get_top_ranked_decade() function
    a. Filters out all other shows that are not in a given decade and returns the number of recommendations the user wants (sorted by rank).

Section 2 - Content Based Recommendations

A Quick Explanation:

Content recommendations require a ‘similarity matrix’. To get a similarity matrix, an attribute matrix is required.

An attribute matrix is a matrix with all of the shows as rows and all of the genres/decades as columns. If a show belongs to a certain genre or decade they will have a ‘1’ in the cell, ‘0’ otherwise.

Here is an example row:

	Comedy	Horror	Vampire	1990s	2000s	2010s
Show 1	0	1	1	0	0	1

So this a vampire horror show from the 2010s.

Taking the dot product of the **whole** attribute matrix with its own transpose will return a similarity matrix.

A similarity matrix is a matrix where the shows are the rows AND the shows are the columns. The numbers in each cell represent how similar two shows are to each other.

Example row:

	Show 1	Show 2	Show 3	Show 4	Show 5
Show 1	3	2	3	2	0
Show 2	2	9	4	7	6
Show 3	3	4	8	6	5
Show 4	2	7	6	6	0
Show 5	0	6	5	0	7

Taking the shows with the higher numbers in each column will give similar show recommendations.

So the steps for the sections are:

1. Create an attribute matrix by indexing the animes_df
2. Create a similarity matrix by taking the dot product of the attributes matrix with the transpose of itself
3. Create a find_similar_shows() function that searches for an anime in the similarity matrix and returns the shows with the highest numbers in the selected row.

Section 3 - Collaborative Based Recommendations

Another quick explanation:

This section requires a user-item matrix to perform the machine learning algorithm. A user-item matrix is a matrix where all of the users (who have left a review of an anime) are the rows and all of the shows are the columns. The ratings a user gives for a show fills the cells. Ratings are from 1-10.

An example row:

	Show 1	Show 2	Show 3	Show 4
User 1	8	NaN	5	7

User 2	NaN	NaN	NaN	6
User 3	6	9	4	NaN

User 1 didn’t rate show 2 which is why it’s labeled as NaN. With a matrix like this, a FunkSVD algorithm can predict what a user 1 might rate show 2. Or user 2 would rate the first 3 shows.

The steps for this function:

1. Separate reviews data into a training and test set
2. Create a FunkSVD algorithm and train the data
3. Hyperparameter tuning
4. Make predictions
5. Prediction validation

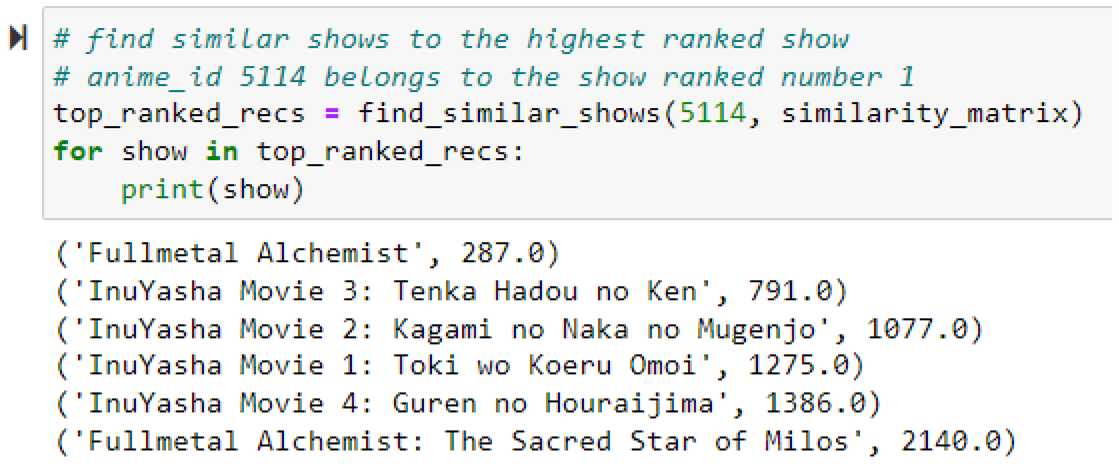
Refinement

Section 1 - Knowledge Based Recommendations

This section did not require any refinement. All returned recommendations are 100% accurate and only required basic pandas dataframe manipulation.

Section 2 - Content Based Recommendation

The function to find similar shows needed some improvement. When passing in the top ranked show (anime id 5114). Only 6 shows were returned. 4 of them were movies from the same franchise. The top recommendation was the previous season and the last recommendation was the movie to the input’s series. These were not good recommendations.



```
# find similar shows to the highest ranked show
# anime_id 5114 belongs to the show ranked number 1
top_ranked_recs = find_similar_shows(5114, similarity_matrix)
for show in top_ranked_recs:
    print(show)
```

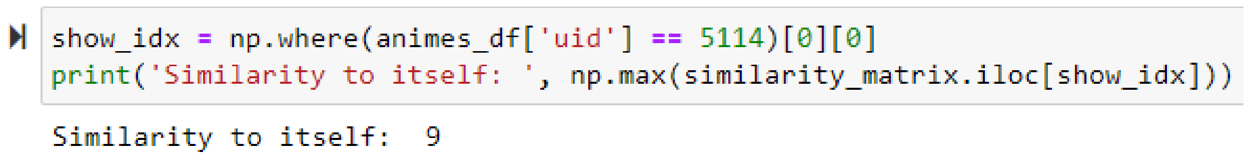
```
('Fullmetal Alchemist', 287.0)
('InuYasha Movie 3: Tenka Hadou no Ken', 791.0)
('InuYasha Movie 2: Kagami no Naka no Mugenjo', 1077.0)
('InuYasha Movie 1: Toki wo Koeru Omoi', 1275.0)
('InuYasha Movie 4: Guren no Houraijima', 1386.0)
('Fullmetal Alchemist: The Sacred Star of Milos', 2140.0)
```

The source of the problem was the following line of code.

```
# find other shows that are similar to the one passed in as an arg
similar_idxs = np.where(similarity_matrix.iloc[show_idx] > np.max(similarity_matrix.iloc[show_idx]-2))[0]
```

This line of code finds the row associated with the show we want to find recs for IN the similarity matrix. It finds the max number in the row (which as seen from the previous section Implement), is the similarity of the show compared to itself. It takes that max number and subtracts two. Any show with a similarity number bigger than that is added as a recommendation.

For this example:
The top ranked show has a similarity number 9 with itself.



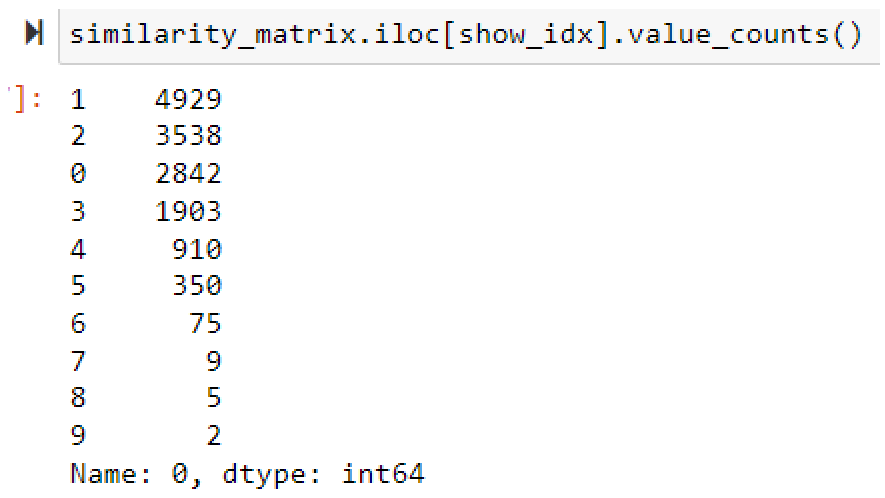
```
show_idx = np.where(animes_df['uid'] == 5114)[0][0]
print('Similarity to itself: ', np.max(similarity_matrix.iloc[show_idx]))
```

```
Similarity to itself:  9
```

The code below will get all shows with a similarity number that is greater than (9-2)

```
# find other shows that are similar to the one passed in as an arg
similar_idxs = np.where(similarity_matrix.iloc[show_idx] > np.max(similarity_matrix.iloc[show_idx]-2))[0]
```

For this show, only 7 recommendations are possible. All of those who have an 8 and 9. But one of those 9's is the show 5114 itself. So that has to be removed.



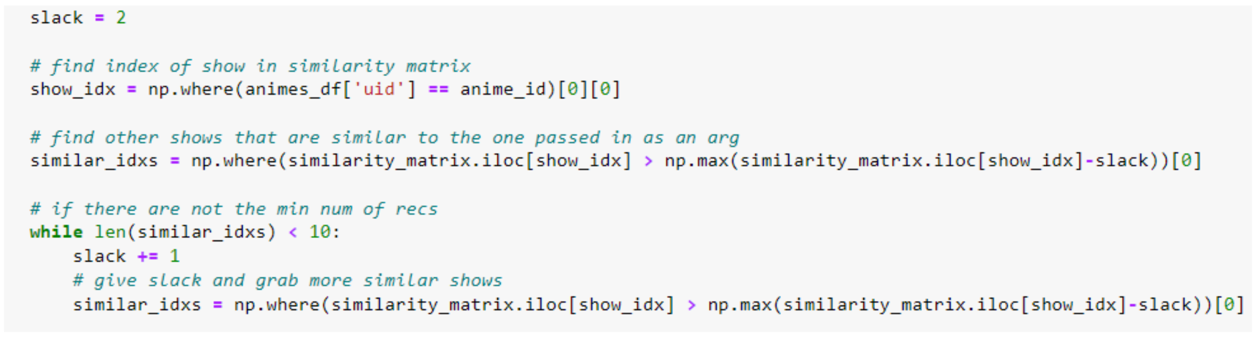
```
similarity_matrix.iloc[show_idx].value_counts()
```

```
]: 1    4929
   2    3538
   0    2842
   3    1903
   4     910
   5     350
   6      75
   7       9
   8       5
   9       2
   Name: 0, dtype: int64
```

Only returning shows with similarity numbers 8-9 fulfills the requirements of the function but it is not satisfying to the user. Shows with a 6 or a 7 could be just as appealing. So instead of subtracting 2 from the max number in a row, maybe subtracting 3 or 4 would lead to good recommendations.

This only works when the max number is high. If the max number in a row was 2, then subtracting two already would make it so that any show regardless how similar it was would return as a recommendation.
The solution I came to was adding 'slack' to the number I subtracted by. The slack variable starts off as 2. If less than 10 recommendations come back, then the slack is increased by 1 until 10 recommendations are returned.

For this example, (9-2) only returned 6 recommendations. So the slack would increase by 1 and the function will find all shows that have a similarity score greater than (9-3)=6. As you can see above, that is everything with a 7-9, or 16 recommendations.



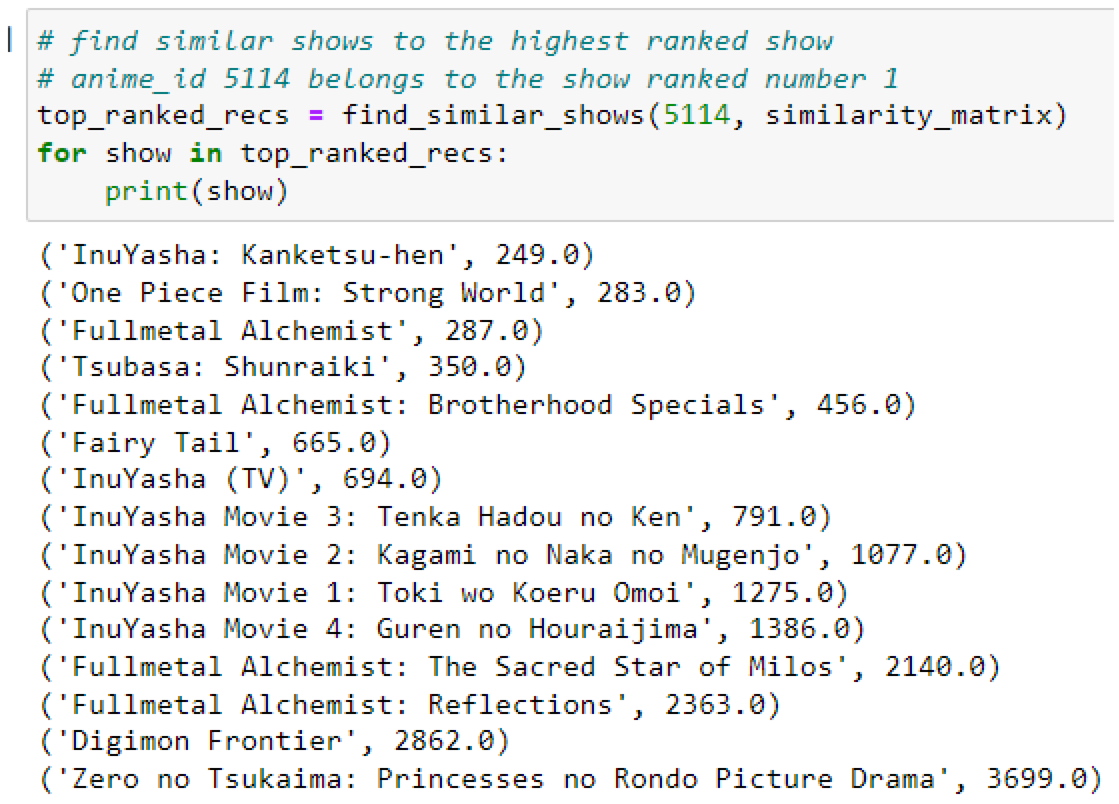
```
slack = 2

# find index of show in similarity matrix
show_idx = np.where(animes_df['uid'] == anime_id)[0][0]

# find other shows that are similar to the one passed in as an arg
similar_idxs = np.where(similarity_matrix.iloc[show_idx] > np.max(similarity_matrix.iloc[show_idx]-slack))[0]

# if there are not the min num of recs
while len(similar_idxs) < 10:
    slack += 1
    # give slack and grab more similar shows
    similar_idxs = np.where(similarity_matrix.iloc[show_idx] > np.max(similarity_matrix.iloc[show_idx]-slack))[0]
```

The improved function from above will result in the following output (title, rank).



```
# find similar shows to the highest ranked show
# anime_id 5114 belongs to the show ranked number 1
top_ranked_recs = find_similar_shows(5114, similarity_matrix)
for show in top_ranked_recs:
    print(show)
```

```
('InuYasha: Kanketsu-hen', 249.0)
('One Piece Film: Strong World', 283.0)
('Fullmetal Alchemist', 287.0)
('Tsubasa: Shunraiki', 350.0)
('Fullmetal Alchemist: Brotherhood Specials', 456.0)
('Fairy Tail', 665.0)
('InuYasha (TV)', 694.0)
('InuYasha Movie 3: Tenka Hadou no Ken', 791.0)
('InuYasha Movie 2: Kagami no Naka no Mugenjo', 1077.0)
('InuYasha Movie 1: Toki wo Koeru Omoi', 1275.0)
('InuYasha Movie 4: Guren no Houraijima', 1386.0)
('Fullmetal Alchemist: The Sacred Star of Milos', 2140.0)
('Fullmetal Alchemist: Reflections', 2363.0)
('Digimon Frontier', 2862.0)
('Zero no Tsukaima: Princesses no Rondo Picture Drama', 3699.0)
```

The recommendations are not based on rank but on similar attributes.

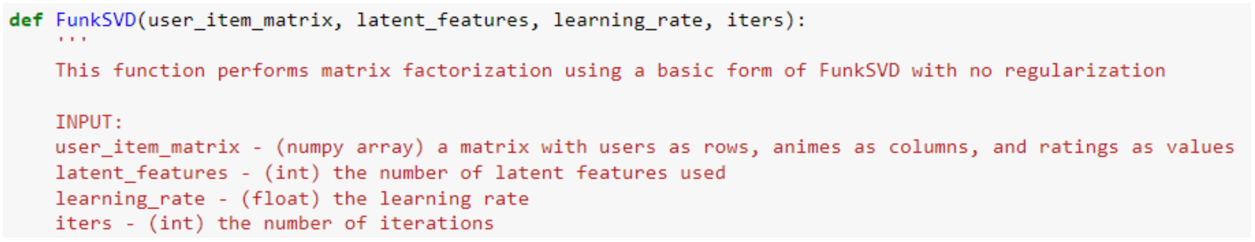
Section 3 - Collaborative Filtering

Refinement for this section was done on the FunkSVD algorithm. Full details can be seen in the following section: Model Evaluation and Validation.

Results

Model Evaluation and Validation

The model takes 4 inputs. The first is the user-item matrix and the other three are parameters to help the model learn and train on the data.



```
def FunkSVD(user_item_matrix, latent_features, learning_rate, iters):
    '''
    This function performs matrix factorization using a basic form of FunkSVD with no regularization

    INPUT:
    user_item_matrix - (numpy array) a matrix with users as rows, animes as columns, and ratings as values
    latent_features - (int) the number of latent features used
    learning_rate - (float) the learning rate
    iters - (int) the number of iterations
```

Latent features are features that can not be directly observed through data but are features that can be inferred based on relationships that occur. For example, a latent feature could be - a show is about samurais. Or a show is about romance. Maybe a show contains a certain character or actress. Finding the relationships between users, shows and latent factors are important for the FunkSVD algorithm.

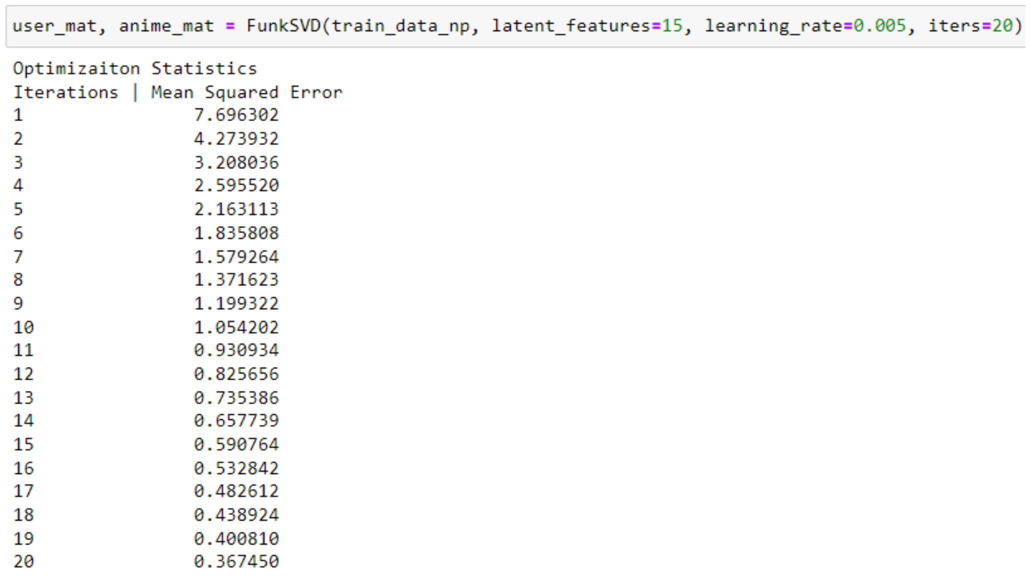
The latent_features variable is the number of latent features that will be observed.

The learning rate is a very small number that is used in the algorithm to help update values. Typically the number is less than one.

The number of iterations is the number of times the algorithm goes through the user_item_matrix to update values.

First iteration

Running FunkSVD takes some time so I put small numbers into the parameters to see how the algorithm would perform. I could tune the parameters after I got a sense of how the parameters would affect the algorithm.



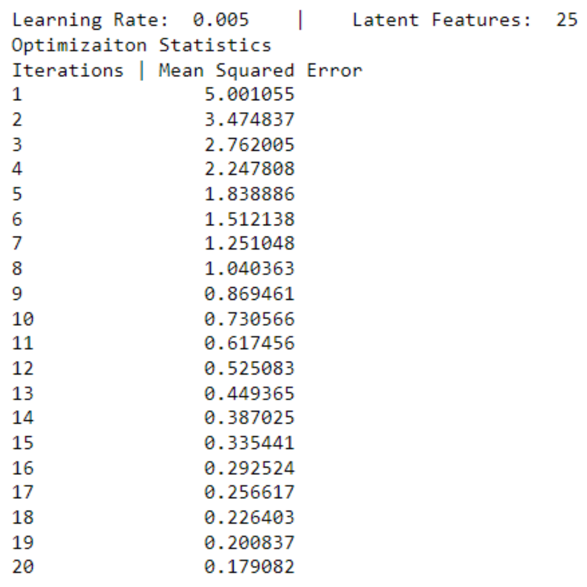
```
user_mat, anime_mat = FunkSVD(train_data_np, latent_features=15, learning_rate=0.005, iters=20)
```

```
Optimizaiton Statistics
Iterations | Mean Squared Error
1		7.696302
2		4.273932
3		3.208036
4		2.595520
5		2.163113
6		1.835808
7		1.579264
8		1.371623
9		1.199322
10		1.054202
11		0.930934
12		0.825656
13		0.735386
14		0.657739
15		0.590764
16		0.532842
17		0.482612
18		0.438924
19		0.400810
20		0.367450
```

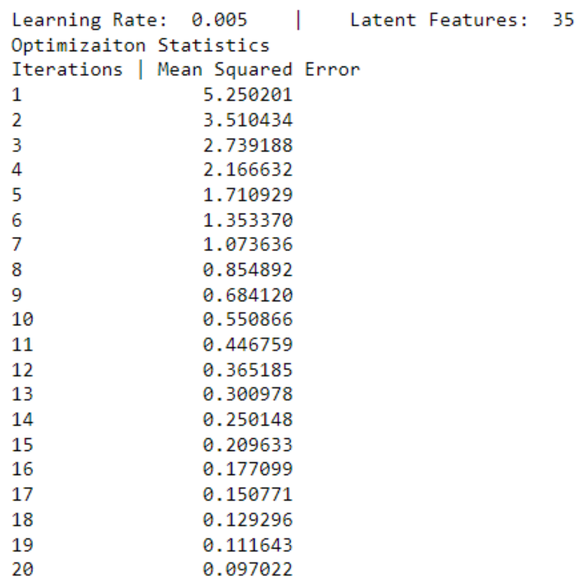
The goal, when measuring by mean squared error, is to minimize the value. The closer to 0 the MSE is, the more accurate the predictions. The first iteration was quite accurate only after 20 iterations. With such a big dataset with a lot of anime shows, I was sure the algorithm could find more relationships in the data. So increasing the number of latent features in the next iterations should result in a lower MSE. It only took 11 iterations to get the MSE below 1 and by the end of the 20th iteration the MSE was below 0.4. The learning rate seemed like a good foundation so I stuck with 0.005 for the time being.

Second Iteration

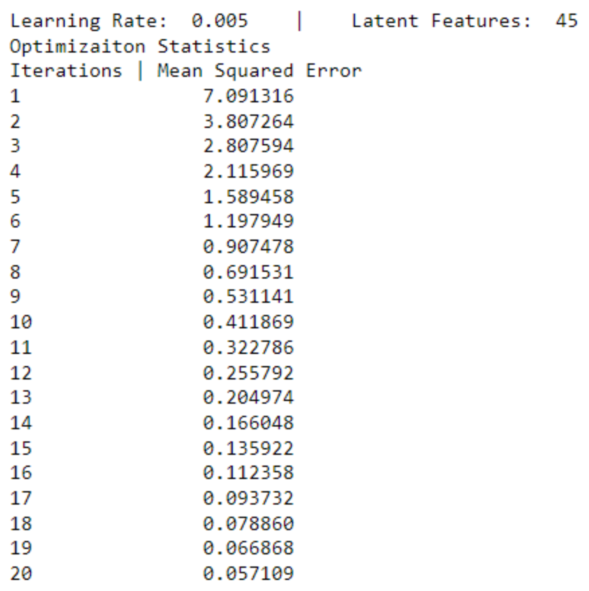
The second iteration I tested only the number of latent features to see their impact on the algorithm.



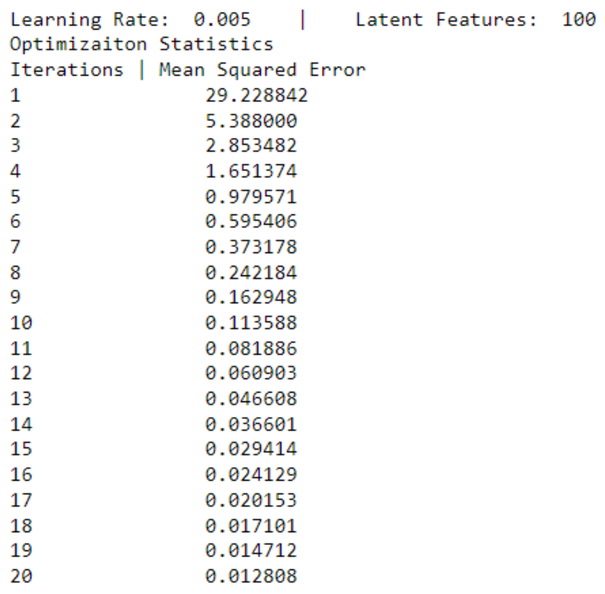
```
Learning Rate:  0.005    |     Latent Features:  25
Optimizaiton Statistics
Iterations | Mean Squared Error
1               5.001055
2               3.474837
3               2.762005
4               2.247808
5               1.838886
6               1.512138
7               1.251048
8               1.040363
9               0.869461
10              0.730566
11              0.617456
12              0.525083
13              0.449365
14              0.387025
15              0.335441
16              0.292524
17              0.256617
18              0.226403
19              0.200837
20              0.179082
```



```
Learning Rate:  0.005    |     Latent Features:  35
Optimizaiton Statistics
Iterations | Mean Squared Error
1               5.250201
2               3.510434
3               2.739188
4               2.166632
5               1.710929
6               1.353370
7               1.073636
8               0.854892
9               0.684120
10              0.550866
11              0.446759
12              0.365185
13              0.300978
14              0.250148
15              0.209633
16              0.177099
17              0.150771
18              0.129296
19              0.111643
20              0.097022
```



```
Learning Rate:  0.005    |     Latent Features:  45
Optimizaiton Statistics
Iterations | Mean Squared Error
1               7.091316
2               3.807264
3               2.807594
4               2.115969
5               1.589458
6               1.197949
7               0.907478
8               0.691531
9               0.531141
10              0.411869
11              0.322786
12              0.255792
13              0.204974
14              0.166048
15              0.135922
16              0.112358
17              0.093732
18              0.078860
19              0.066868
20              0.057109
```

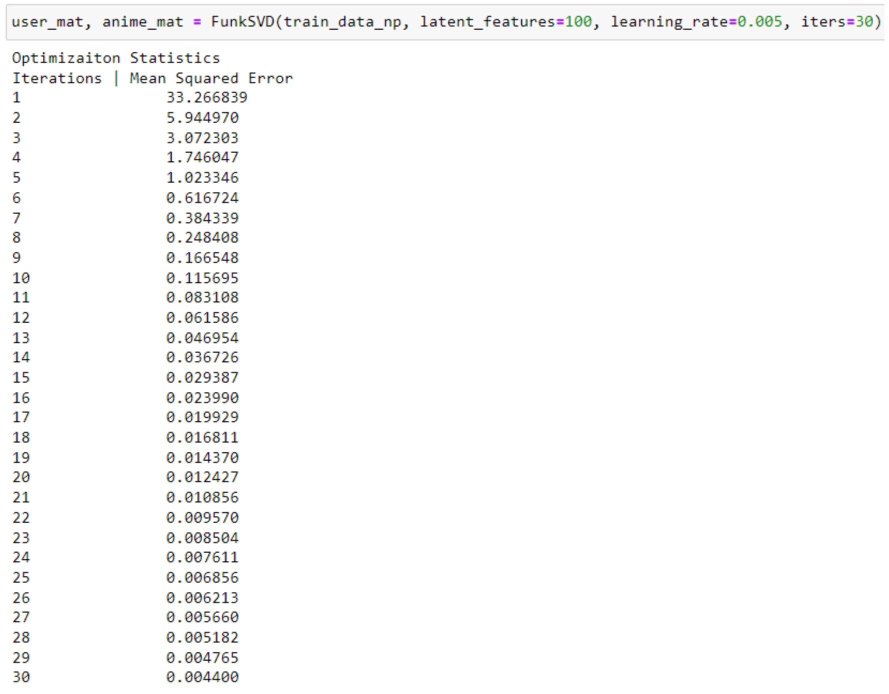


```
Learning Rate:  0.005    |     Latent Features:  100
Optimizaiton Statistics
Iterations | Mean Squared Error
1               29.228842
2               5.388000
3               2.853482
4               1.651374
5               0.979571
6               0.595406
7               0.373178
8               0.242184
9               0.162948
10              0.113588
11              0.081886
12              0.060903
13              0.046608
14              0.036601
15              0.029414
16              0.024129
17              0.020153
18              0.017101
19              0.014712
20              0.012808
```

The more latent features that are used, the more accurate the model becomes. I was still satisfied with the learning rate so I chose to keep it as the final value. A MSE of 0.0128 was a lot better than the 0.36 value I had when only using 15 latent features.

Final Parameters

The last test I did was to see how minimal the MSE would become if I let the algorithm run for 10 more iterations. The final MSE was 0.0044 and I'm sure it would have gotten a lot smaller if the algorithm was given more than 50 iterations. For time purposes I chose to keep the iterations at 30. The algorithm itself would take around 30 minutes to finish each run.



```
user_mat, anime_mat = FunkSVD(train_data_np, latent_features=100, learning_rate=0.005, iters=30)

Optimizaiton Statistics
Iterations | Mean Squared Error
1               33.266839
2               5.944970
3               3.072303
4               1.746047
5               1.023346
6               0.616724
7               0.384339
8               0.248408
9               0.166548
10              0.115695
11              0.083108
12              0.061586
13              0.046954
14              0.036726
15              0.029387
16              0.023990
17              0.019929
18              0.016811
19              0.014370
20              0.012427
21              0.010856
22              0.009570
23              0.008504
24              0.007611
25              0.006856
26              0.006213
27              0.005660
28              0.005182
29              0.004765
30              0.004400
```

Justification

Latent Features - I chose to keep the latent features at 100 because the results after 20-30 iterations were clearly better than any of the other options I chose. With so many shows only sharing 40 genres, there were many relationships in the data that the FunkSVD algorithm could make. 100 latent features were simply the best performing.

Learning Rate - the learning rate was the definition of trial and error. Anything other than 0.005 resulted in a “RuntimeWarning: overflow encountered in double_scalars”. There were very few answers that helped with solving the issue so I stuck with 0.005 as the learning rate. There is a

chance that FunkSVD was computing a number outside the range of a numpy double. See below for details:
https://stackoverflow.com/questions/43405777/what-are-the-causes-of-overflow-encountered-in-double-scalars-besides-division-b

Iterations - The final number of iterations used was 30. 20 was used for testing which number of latent features resulted in a lower MSE. I decided on 30 due to time constraints. FunkSVD on this data runs a little slow. I did not want the algorithm to run for more than 20 minutes but each iteration takes about a minute to complete. With 30 iterations the MSE got as low as 0.0044 which is sufficient for this project. No more time should have been spent waiting for it to run.

Conclusion

Reflection

The dataset taken from Kaggle made knowledge and content-based recommendations straightforward to implement. I did not have to spend hours cleaning the data and preparing for the implementation. The first two types of recommendations were quick and easy.

The difficulty of this project was in hyperparameter tuning for the FunkSVD model. The number of latent features, the learning rate and the number of iterations all had to be chosen by trial and error. All I could do was plug in numbers that I thought would return good results and go from there. The learning rate, if increased or decreased from the final value, would result in the algorithm running and learning slower. After a few iterations a RuntimeWarning: overflow error would be thrown and all MSE values coming back were nan. I could only test different number of latent features and number of iterations to tune the model.

Overall I think this is a good foundation to build off of. Knowledge and Content based recommendations were of high quality but the collaborative recommendations could use some fine tuning.

Improvement

The best way I could improve this project is to familiarize myself with other forms of evaluation metrics for FunkSVD. I used Mean Squared Error to evaluate the performance of my model but there are a number of different evaluation metrics like Mean Absolute Error, Root Mean Squared Error, Precision, Recall or Mean Reciprocal Rank. Using a different metric or a combination of metrics might give more insight into how well the model actually performs.