MATH 496T Matrix Methods for Recommendation Systems
Final Project

Ainsley Pahljina

May 9, 2020

Introduction

Matrix completion describes the process of inferring missing entries from a partially completed matrix. The utility of matrix completion is broad-reaching, from simple imputation of missing data to large matrix recovery for compressed sensing, image processing or recommendation systems. The problem of matrix completion was glamourised by the 2009 Netflix Prize. If one considers a comprehensive matrix of users and movies, matrix completion may be used to predict a rating for a given user, movie entry. Needless to say, offering a $1 million prize has attracted an influx of research on matrix completion methods. Much of the recent literature is difficult to digest, and there is very limited tangible demonstrations of the applications. This project attempts to present various methods of matrix completion by applying them to a movie recommendation data set. This has been inspired by an attempt by Nick Becker (https://beckernick.github.io/matrix-factorization-recommender/) that used the process of matrix factorisation by singular value decomposition to generate a recommendation matrix. The paper will have a computational focus, detailing the application of these methods and their intuition rather than providing a mathematically rigorous justification.

Data

The data has been extracted from a movie recommendation platform that has collated their data for education and research purposes (https://grouplens.org/datasets/movielens/). For computational ease, the data extracted is a smaller subset of what is available - it contains the ratings of 610 users for 9724 movies. The data set split between movies and ratings can be formatted to our desired data matrix, as illustrated below:

[66]:
```
movies.head()
```

[66]:
```
   movieId                               title  \
0        1                    Toy Story (1995)
1        2                      Jumanji (1995)
2        3             Grumpier Old Men (1995)

                                        genres
0  Adventure|Animation|Children|Comedy|Fantasy
1                   Adventure|Children|Fantasy
2                               Comedy|Romance
```

[67]:
```
ratings.head()
```

[67]:
```
   userId  movieId  rating  timestamp
0       1        1     4.0  964982703
1       1        3     4.0  964981247
2       1        6     4.0  964982224
```

[68]:
```
# form our recommendation matrix
R_df = ratings.pivot(index = 'userId', columns ='movieId', values = 'rating').
 ↪fillna(0)
R_df.head()
```

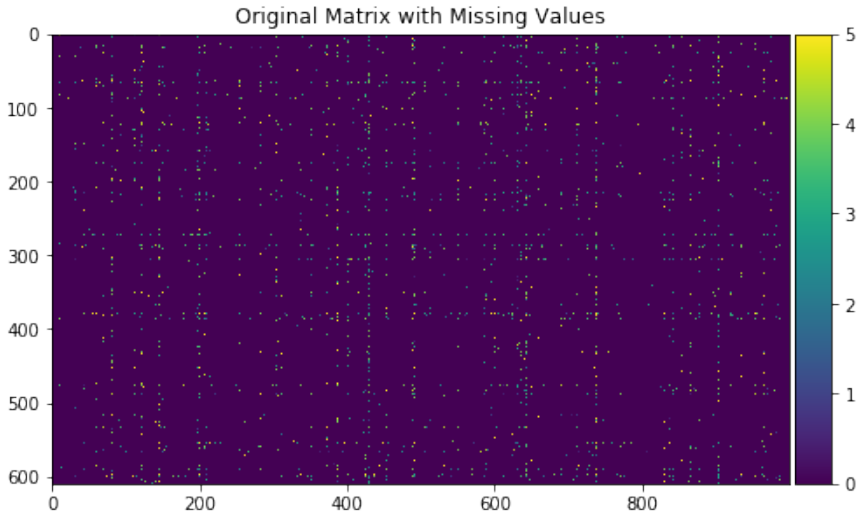
[68]:
```
movieId  1       2       3       4       5       6       7       8       \
userId
1          4.0     0.0     4.0     0.0     0.0     4.0     0.0     0.0
2          0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
3          0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
4          0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
5          4.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0

[5 rows x 9724 columns]
```

Even in this modified demonstration, the matrix we are handling is large. Moreover, by counting the non-missing ratings we see that actual observations only account for 1.7% of the matrix. We can see this visually, the missing values as zero entries.

[69]:
```
# visualise original matrix
plt.figure(figsize = (8,8))
im = plt.imshow(R_df.T.sample(1000).T)
ax =plt.gca()
plt.title("Original Matrix with Missing Values")
divider = make_axes_locatable(ax) # fix colour bar
cax = divider.append_axes("right", size="5%", pad=0.05)
plt.colorbar(im, cax=cax)
```

[69]:
```
<matplotlib.colorbar.Colorbar at 0x127fe8a10>
```



[70]:
```
# function to count non-missing values
def countval(df):
    val = 0
    for i in range(0,610):
        for j in range(0,9724):
            if (df.iloc[i,j] != 0):
                val += 1
    return val
```

[8]:
```
# very slow to run
countval(R_df)
```

[8]: 100836

Testing

Now, as an extension on Becker's recommendation system, this project will attempt to test each methodology. Thus, in disaggregating the data into a training and testing set we can compute an error metric. In randomly selecting 30% of the data to be the test set, any entry that may have had a value attributed to them will be set to missing.

[71]:
```
# create our testing set
test = R_df
missingvals = []

# randomly select 30% of the data to be removed for testing purposes
ix = [(row, col) for row in range(R_df.shape[0]) for col in range(R_df.shape[1])]
for row, col in random.sample(ix, int(round(.3*len(ix)))):
    if (R_df.iloc[row,col] != 0):
        missingvals.append((row,col))
        test.iat[row,col] = 0
```

[72]:
```
len(missingvals) # the number of ratings removed for testing purposes
```

[72]: 30335

Methods and Implementation

Singular Value Decomposition (SVD)

The SVD decomposes a matrix into the best, lower rank approximation of the original matrix. Due to the under-determined nature of our matrix, the matrix is assumed to be low rank. Although strictly a method of matrix factorisation, the truncated SVD yields a low rank, completed matrix representative of our observed data. In obtaining our predicted matrix, $P = U\Sigma V^T$, U captures

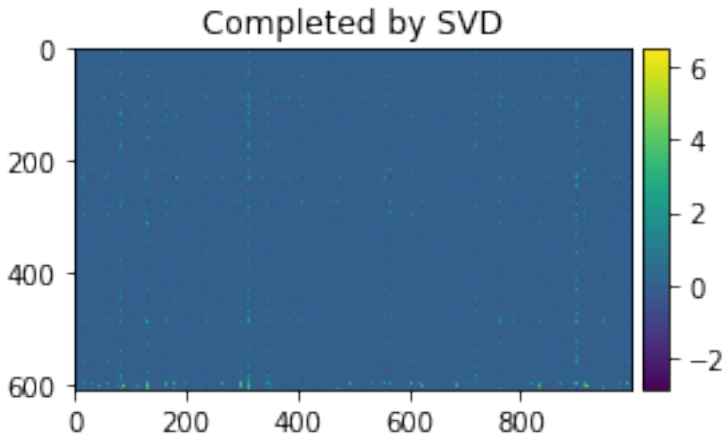
the users 'features', V^T captures the films 'features' and the singular values encapsulated in Σ provide weightings. By increasing the value of k, the algorithm will inherently capture more relationships between users and films, at the expense of the rank and computational time. This may be especially valuable in completion problems where the rank of the underlying matrix is known. This was the method utilised by Becker, thus the code below illustrates the process of implementation using scipy. Moreover, a visualisation of the completed matrix is included for a more intuitive understanding of the method. Although the scale has been changed, the relative values are still meaningful.

[73]:
```
# normalise data matrix for SVD fitting
T = test.values
user_ratings_mean = np.mean(T, axis = 1)
test_demeaned = T - user_ratings_mean.reshape(-1, 1)
```

[74]:
```
# truncated SVD - as demonstrated
from scipy.sparse.linalg import svds

start = time.time()
U, sigma, Vt = svds(test_demeaned, k = 30)
elapsed_time_SVD = (time.time() - start)
```

[75]:
```
sigma = np.diag(sigma)
all_user_predicted_ratings = np.dot(np.dot(U, sigma), Vt) + user_ratings_mean.
 ↪reshape(-1, 1)
SVD_df = pd.DataFrame(all_user_predicted_ratings, columns = R_df.columns)
```



For the remaining completion algorithms, the missing values are required to be nan instead of zero which may be interpreted as observed. The following function rather inefficiently sets these missing values as nan.

[79]:
```
# slow to run
# function to convert zero values to nan (needed for some algorithms)
def zerotonan(df):
    for i in range(df.shape[0]):
        for j in range(df.shape[1]):
            if(df.iloc[i,j] == 0):
                df.iat[i,j] = np.nan
    return(df)
test = zerotonan(test)
```

Simple Imputation

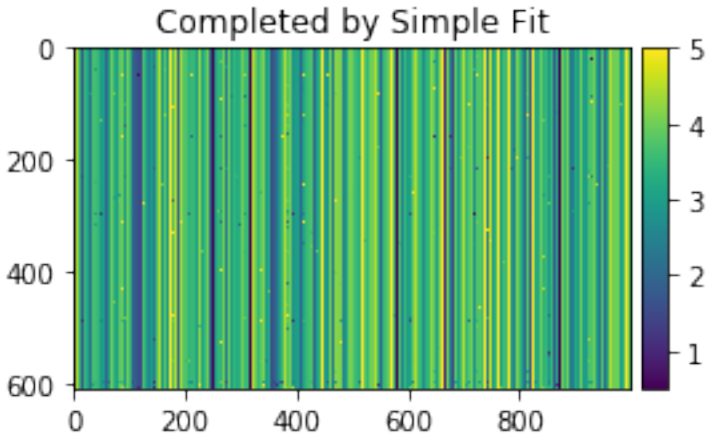
This method attributes a value to a missing entry by computing the average of the existing entries its columns. From the visualisation, we can see the column values have been meaned giving the appearance of stripes. Looking closely, the initial observations are spotted throughout the image. Although there is nothing sophisticated about this method it has been included for comparison with the other methods. The performance and outcomes of simple imputation should not outcompete the other methodologies.

[80]:
```
# simple imputer
from sklearn.impute import SimpleImputer
simp = SimpleImputer(missing_values=np.nan, strategy = "mean")
```

[81]:
```
start = time.time()
simple_fit = simp.fit_transform(test)
elapsed_time_simp = (time.time() - start)

simple_fit = pd.DataFrame(simple_fit)
```

[82]:
```
# reassigning column names for merging purposes
simple_fit.index.name = "userId"
simple_fit.T.index.name = "movieId"
```



For the remaining algorithms, the library **fancyimpute** has been used (https://github.com/iskandr/fancyimpute).

[83]:
```
from fancyimpute import SoftImpute, KNN
```

K-Nearest Neighbours (KNN)

The KNN algorithm uses a defined notion of closeness to allocate a value to a missing point based on its neighbours. This works intuitively - the algorithm could be used to attribute a missing rating for a given user and movie based on its neighbours (i.e. the ratings for this movie by other users that have similar preferences to this user). In this data set the movies are ordered by date, so the notion of closeness column wise is probably less indicative of a given rating unless a specific user rates all movies in a similar way or favours a certain era of films. The algorithm in **fancyimpute** considers only the k-nearest rows.

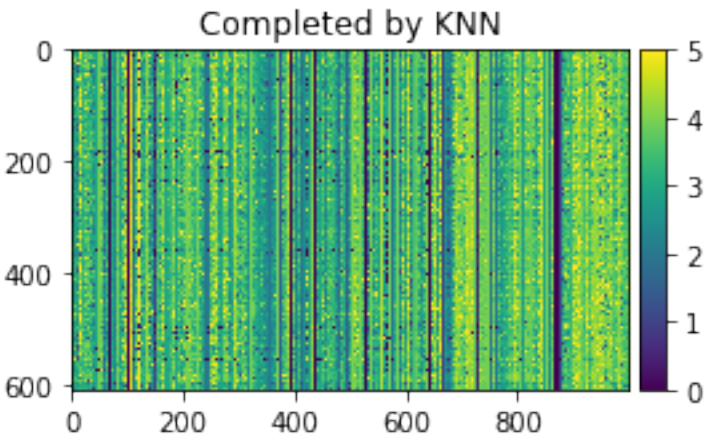
So mathematically, let's consider a given missing entry M_{ij} which belongs to a user that has rated some subset of movies ($j \in S$). Now, we calculate the mean square difference for the remaining rows ($k \neq i$) and columns $\bar{j} = \{j \in S : M_{kj} \neq 0\}$ that have non-missing entries:

$$\frac{\sum_{\bar{j}}(M_{kj} - M_{ij})^2}{|\,\bar{j}\,|}$$

Now using this metric of closeness, we impute the rating associated with the most similar user. So the algorithm finds a user with the most similar movie taste, and imputes their rating for a given movie.

This seems logical, however this relies on a certain level of density to effectively inform similarity between users. Where ratings account for just 1.7% of the data matrix, the amount of overlap between users is likely to be minimal. For some films, the algorithm failed to impute ratings as the information was so scarce and thus the rating was set to zero. This is visible in the image below and must be kept in mind later when accuracy is considered. This may be overcome by considering more neighbours, at the expense of computation time.

[84]:
```
start = time.time()
X_filled_knn = KNN(k=10).fit_transform(test.values)
elapsed_time_KNN = (time.time() - start)
```



Nuclear Norm Minimisation

As discussed, the matrix we are working with is highly under-determined. To overcome this, the matrix is assumed to be low rank. By minimising the rank, the degrees of freedom are restricted, and thus the training error is similarly minimised. So the goal is to find a low rank complete matrix, that aligns with the pre-existing observations. Mathematically, we consider that for partially observed $M \in \mathbb{R}^{n \times n}$ and a specified small rank r, we wish to find $X \in \mathbb{R}^{n \times n}$ such that we minimise:

$$\sum_{Observed(i,j)} (X_{ij} - M_{ij})^2 \text{ (subject to rank(X) = r)}$$

Now this rank constraint renders the problem non-convex, and computationally difficult. A relaxation of this problem instead bounds the nuclear norm. Now, let's first define the nuclear norm as the sum of the singular values of X:

$$|| \ X \ ||_* = \sum_j \lambda_j(X)$$

Now we consider rewriting the stated criterion by considering the following projection onto the observed entries:

$$P_\Omega(M)_{ij} = \begin{cases} M_{ij} & \text{(i,j) is observed} \\ 0 & \text{(i,j) is missing} \end{cases}$$

Equality of $P_\Omega(M)$ and $P_\Omega(X)$, enforces that the observed entries do not alter in the completed matrix. However, to account for noise in the data we instead consider $|| \ P_\Omega(M) - P_\Omega(X) \ ||_F \leq \delta$, for some specified, but small delta parameter. This condition is advantageous in such a sparse matrix, as it is easier to uphold a smaller number of pre-existing observations. Although convex, this algorithm is highly expensive in practise. Whilst it was included in the **fancyimpute** library, it would not be able to handle the size of our data set. So now we consider an alternative to solving this minimisation that relies on iteration.

softImpute

If we return to a similarly constructed optimisation problem, to minimise:

$$\frac{1}{2} \mid\mid P_\Omega(M) - P_\Omega(X) \mid\mid_F^2 + \lambda \mid\mid X \mid\mid_*$$

A solution to this is the soft-thresholded SVD

$$S_\lambda(X) := U.\,\text{diag}\{(\sigma_1 - \lambda)+, ..., (\sigma_m - \lambda)\}.V^T$$

The key idea is that iterative soft thresholding converges to an optimal, low rank solution, with many of the diagonal entries converging to zero. Let's consider an overview of the algorithm:

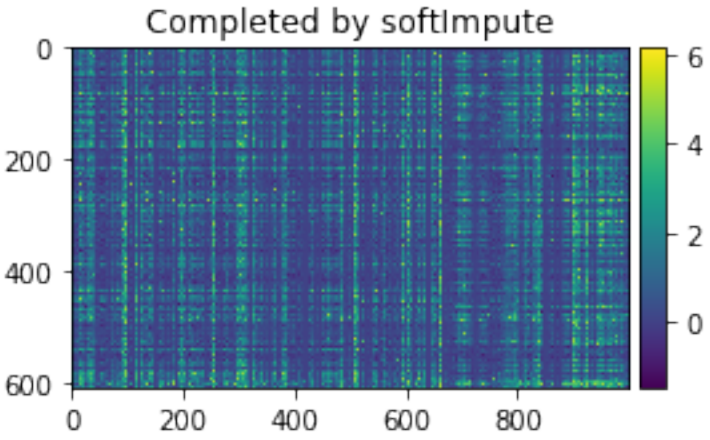
1. Initialise $X^{old} = 0$, and generate a decreasing series of λ values, with $\lambda_0 = \lambda_{max}(P_\Omega(M))$.
2. For each λ, we iterate over the following:
   - $Z^{new} = S_\lambda(P_\Omega(M) + P_\Omega^\perp(X))$
   - Z^{new} replaces Z^{old} and corresponds to the solution for the given iteration.

This favours the sparse and desirable low rank structure of our matrix. The number of iterations of the algorithm is specified by construction. The completed matrix obtained here is a product of 100 iterations, and the rank was reduced to 141. For some columns however, the algorithm defaulted to imputing identical values and it was unclear why. As an extension of this paper, this algorithm could perhaps be implemented outside of the **fancyimpute** library to understand the source of this. The normalisation of the testing matrix, application of the softImpute algorithm and visualisation of the completed matrix is illustrated below. It is also noted that the matrix was unnormalised following this to ensure the values themselves remained meaningful as ratings.

[113]:
```
import pandas as pd
from sklearn import preprocessing
#normalise test set
x = test.values
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
test_norm = pd.DataFrame(x_scaled)
```

[114]:
```
# applying the soft impute algorithm
start = time.time()
df_filled_softimpute = SoftImpute().fit_transform(test_norm.values)
elapsed_time_SI = (time.time() - start)
```

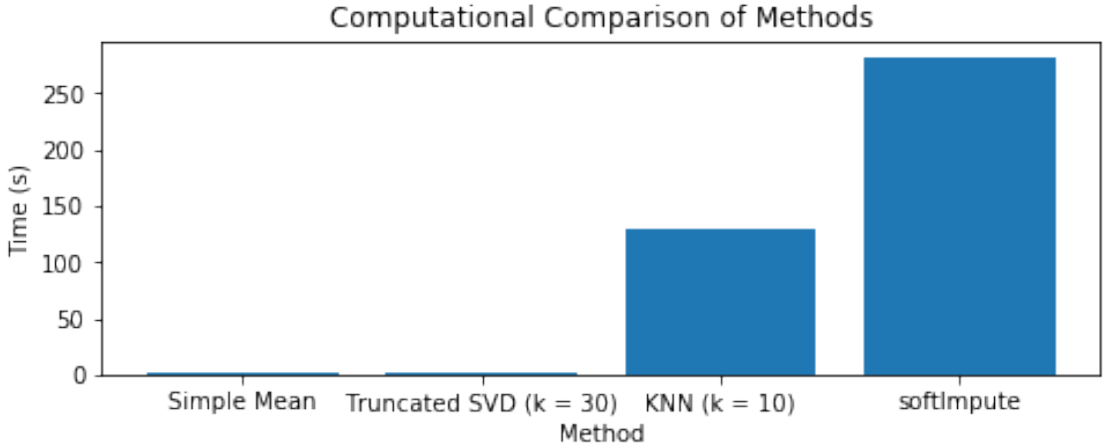
[132]:
```
<matplotlib.colorbar.Colorbar at 0x1aade52d10>
```



Results

Complexity

In considering the computational time of each method, it is clear that the KNN and soft-impute algorithms are significantly more expensive. The involvement of these method depends on the user specified neighbour (k-value) and number of iterations, respectively.



Error

Now, in considering the performance of each of these methods on our testing set we calculate the mean squared error against the actual values. This function is defined below.

[89]:
```
# defining error function
def error(df, missingvals, completedmatrix):
    count = 0
    sm = 0
    for (i,j) in missingvals:

            if ((df.iloc[i,j] - completedmatrix.iloc[i,j]) == (df.iloc[i,j] -␣
 ↪completedmatrix.iloc[i,j])): #accounting for nan values
                sm = sm + (df.iloc[i,j] - completedmatrix.iloc[i,j])**2
                count = count + 1
    MSE = sm/count
    print("MSE for this method is {}".format(MSE))
```

[40]:
```
error(R_df, missingvals, simple_fit)
```

```
MSE for this method is 1.5601727427787555
```

[61]:
```
error(R_df, missingvals, SVD_df)
```

```
MSE for this method is 9.755283732297592
```

[42]:
```
error(R_df, missingvals, SI_fit)
```

```
MSE for this method is 1.2938058847771556
```

[43]:
```
error(R_df, missingvals, KNN_completed)
```

```
MSE for this method is 1.1288695869877048
```

The softImpute algorithm and KNN only performed marginally better than the basic mean column filler. This is unexpected, and potentially reflects the flaws in the algorithms implementation. Moreover, in computing the errors for several different randomised test sets there was a fair amount of variability indicating that the error may be skewed by a few problematic entries. This may require further investigation. Furthermore, the calculation of the error term is somewhat meaningless in regards to the SVD computation, as shown above the values do not pertain to the original rating scale. Perhaps a better indication of error in this respect would be to consider each film as ranked by the users ratings, and how the predicted rank aligns to the actual ranking. However, as the ratings are somewhat discrete, a ranking is less meaningful for the original data.

Recommendations

By adapting a function from Becker's implementation, we are able to extract the top predictions for a specified user, and compare this to their top rated films. This is obviously a subjective assessment of quality, however it was interesting to ascertain any bias in the algorithms.

```
User 24 has already rated 110 movies.
Recommending the highest 5 predicted ratings movies not already rated.
```

Top Rated Movies

[91]:

Heat (1995)	Action|Crime|Thriller
Forrest Gump (1994)	Comedy|Drama|Romance|War
The Count of Monte Cristo (2002)	Action|Adventure|Drama|Thriller
Laputa: Castle in the Sky (Tenkû no shiro Rapy...	Action|Adventure|Animation|Children
Old Boy (2003)	Mystery|Thriller
Groundhog Day (1993)	Comedy|Fantasy|Romance
Raiders of the Lost Ark (Indiana Jones and the...	Action|Adventure
Silence of the Lambs, The (1991)	Crime|Horror|Thriller
Green Mile, The (1999)	Crime|Drama
Shawshank Redemption, The (1994)	Crime|Drama

Simple Impute - Top Predictions

[92]:

Monster (2003)	Crime|Drama
Embalmer, The (Imbalsamatore, L') (2002)	Drama
The Magician (1958)	Drama
Laputa: Castle in the Sky (Tenkû no shiro Rapy...	Action|Adventure|Animation|Children
Treasure of the Sierra Madre, The (1948)	Action|Adventure|Drama|Western
Airport '77 (1977)	Drama

SVD - Top Predictions

[93]:

Star Wars: Episode IV - A New Hope (1977)	Action|Adventure|Sci-Fi
Star Wars: Episode V - The Empire Strikes Back...)	Action|Adventure|Sci-Fi
Fight Club (1999)	Action|Crime|Drama|Thriller
American Beauty (1999)	Drama|Romance
Jurassic Park (1993)	Action|Adventure|Sci-Fi|Thriller

KNN - Top Predictions

[94]:

Cat's Meow, The (2002)	Drama|Thriller
Eulogy (2004)	Comedy|Crime|Drama
Rose Tattoo, The (1955)	Drama|Romance
Cellular (2004)	Action|Crime|Drama|Mystery|Thriller
Dennis the Menace (1993)	Comedy

softImpute - Top Predictions

[95]:

Indestructible Man (1956)	Crime|Horror|Sci-Fi
Manhattan Project, The (1986)	Comedy|Sci-Fi|Thriller
Secret Window (2004)	Mystery|Thriller
Dawn of the Dead (1978)	Action|Drama|Horror
Die Hard (1988)	Action|Crime|Thriller

So considering these predictions, the following observations were made:

- The simple imputation would predict movies with the highest rating average across all users that the user has not seen. Judging by the obscurity of the movies suggested, this could potentially be biased by a few movies that have been rated highly only by very few users. More popular movies are subject to the full spectrum of tastes and preferences and thus their average rating may be lower. This conclusion is fairly speculative however.
- The SVD predicts movies that appear to align with the users interests. It was interesting to note the inclusion of two Star Wars movies - one would assume their ratings are dependent, and thus this is reflective of an effective predictor. This method also appears favour popular films.
- The KNN and softImpute algorithm predictions aren't as clearly reflective of popularity or user preference. Again, this may be biased by the erroneous completion of some columns as either sent to zero or attributed a constant value.
- It is noted that many of the recommendations are from a similar era to the users preferences (approximately 90s/2000s). This may be observation bias on my behalf however.

Conclusion

This paper attempts to outline various matrix methods for completion and recommendation and step through their implementation on a tangible data set. The methods detailed including simple mean imputation, truncated SVD, K-nearest neighbours and softImpute represent the full spectrum of sophistication. The SVD method proved to yield the most sensible predictions, however it would be interesting to trial this method varying the k-rank. The KNN approach struggled with the sparsity of our matrix, defaulting many values with insufficient neighbours to zero. Again, trialling this method by considering more neighbours may remedy this. The computational expense deterred this route of exploration. The softImpute algorithm similarly defaulted for some columns, and it would be worthwhile to implement this outside the fancyimpute function to detect how this is occurring. Although this algorithm produced the lowest error on the testing set, it was comparable in scale to the error of the basic mean imputation. In providing a computational overview, this paper necessarily glossed over some of the underpinning mathematical proofs and necessary assumptions. In conclusion, this paper provides a strong overview of methods, implementation and limitations, however there is scope to further explore this area.

References

- *Matrix Factorisation for Movie Recommendations in Python* (Becker, 2016)
  https://beckernick.github.io/matrix-factorization-recommender/
- *Comparing Neighbourhood and Matrix Factorization models in Recommendation Sytems* (Torstensson, 2019)
  http://www.diva-portal.org/smash/get/diva2:1414472/FULLTEXT01.pdf
- *Spectral Regularization Algorithms for Learning Large Incomplete Matrices* (Mazumder, Hastie, Tibshirani, 2010)
  https://web.stanford.edu/~hastie/Papers/mazumder10a.pdf
- *Low Rank Matrix Completion: A Contemporary Survey* (Nguyen, Kim, Shim, 2019)
  https://arxiv.org/abs/1907.11705
- *Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares* (Hastie, Mazumder, Lee, and Zadeh, 2015)
  http://jmlr.org/papers/volume16/hastie15a/hastie15a.pdf
- *fancyimpute* - Python Library https://pypi.org/project/fancyimpute/