Contributions

No.	Name	Description
1	Marcus Kornmann	**Sections:** Task description, Algorithms, Results
2	Jie Ni	**Sections:** Algorithms, Results, Difficulties
3	Heqi Yin	**Sections:** Algorithms, Results, Difficulties
4	Rishabh Agrawal	**Sections:** Algorithms, Results and Comparison, Difficulties

Section 1 – Task description

The goal of our final project was to perform matrix completion on a dataset collected through a survey of students in a class about their favorite restaurants around campus. The dataset was represented as a matrix, with rows representing students and columns representing restaurants. However, half of the data points in the matrix were deleted, creating gaps in the data. A subset of the data can be seen in the following table.

V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15
3	N/A	N/A	3	N/A	3	3	N/A	N/A	3	3	N/A	4	N/A	3
N/A	3	3	N/A	N/A	N/A	3	3	N/A	N/A	3	N/A	5	N/A	3
N/A	N/A	3	3	3	3	3	3	N/A	N/A	N/A	4	3	3	N/A
4	4	N/A	N/A	3	N/A	N/A	3	N/A	2	N/A	5	N/A	4	N/A

To build and evaluate our models we used two datasets: MovieLens and Jester. The MovieLens dataset also is a recommendation dataset. In its original form it has 25 million ratings from 270 thousand users for 62,000 movies. Since this is too large for computational reasons, we use a

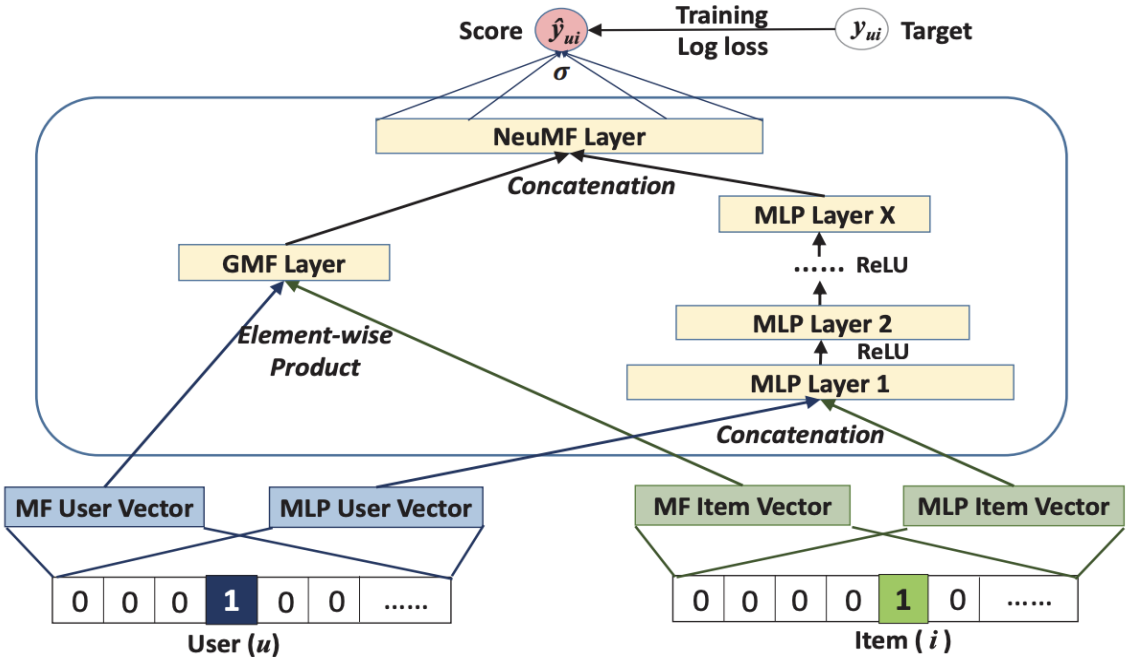
subset of 600 users and 9000 movies. As in the restaurant dataset, each rating is a natural number from one to five.

To use the dataset, we replaced the missing values by the column mean and randomly removed 50% of the data again. These data points are then imputed by our algorithms. Let D be the matrix before we filled missing values by column mean and Y be the result after imputing. We calculated the RMSE using D and Y.

Section 2 – Algorithms

NeuMF (Neural Matrix Factorization)

We have referred the paper - Neural Collaborative Filtering, a technique that combines the power of neural networks with the principles of matrix factorization for collaborative filtering. It replaces the traditional inner product operation in matrix factorization with a neural architecture that can learn complex interactions between user and item features. NMF aims to capture both linear and non-linear relationships, allowing for more accurate and expressive modeling of user-item preferences in recommendation systems. By leveraging the flexibility of neural networks, NMF offers improved performance compared to traditional matrix factorization methods.



To provide more flexibility to the fused model, we allow GMF and MLP to learn separate embeddings, and combine the two models by concatenating their last hidden layer. The formulation of which is given as follows-

$$\phi^{GMF} = \mathbf{p}_u^G \odot \mathbf{q}_i^G,$$
$$\phi^{MLP} = a_L(\mathbf{W}_L^T(a_{L-1}(...a_2(\mathbf{W}_2^T \begin{bmatrix} \mathbf{p}_u^M \\ \mathbf{q}_i^M \end{bmatrix} + \mathbf{b}_2)...)) + \mathbf{b}_L),$$
$$\hat{y}_{ui} = \sigma(\mathbf{h}^T \begin{bmatrix} \phi^{GMF} \\ \phi^{MLP} \end{bmatrix}),$$

KNN

KNN is a common classification and prediction model. In data inputting problems, it is commonly used to investigate and fill in missing values in the dataset.KNN Imputer is used to estimate the missing values by a weighted average of the distances of the K observations closest to the missing values. There are various distance measures available to measure the distance between observations, with Euclidean distance being the most commonly used.

The formula for the Euclidean distance for two points P and Q in 'n' dimensional space:

$$D(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

Where p1,p2,...,pn and q1,q2,...,qn are the coordinates of the points P and Q respectively.

Then the missing values are then filled by the weighted average of the distances

Soft Impute

Soft-Impute is a matrix completion algorithm used to impute missing values in a matrix. It uses the idea that low-rank matrices are good approximations for high-dimensional data. To estimate the low-rank structure, soft-impute uses Singular Value Decomposition. Let $Z \in \mathbb{R}^{m \times n}$ be the matrix we try to complete, $\Omega \subset \{1, ..., m\} \times \{1, ...n\}$, the our goal is to solve

$$\underset{\mathbf{M}}{\text{minimize}} \left\{ \frac{1}{2} \sum_{(i,j) \in \Omega} (z_{ij} - m_{ij})^2 + \lambda \|\mathbf{M}\|_\star \right\}$$

Where $\|\cdot\|_*$ is the nuclear norm. If we have given an observed subset of Ω, we can define a projection function $P_\Omega: \mathbb{R}^{m \times n} \to \mathbb{R}^{m \times n}$ as

$$[\mathcal{P}_\Omega(\mathbf{Z})]_{ij} = \begin{cases} z_{ij} & \text{if } (i,j) \in \Omega \\ 0 & \text{if } (i,j) \notin \Omega, \end{cases}$$

So that we can write this:

$$\sum_{(i,j)\in\Omega}(z_{ij}-m_{ij})^2=\|\mathcal{P}_\Omega(\mathbf{Z})-\mathcal{P}_\Omega(\mathbf{M})\|_{\mathrm{F}}^2.$$

Finally, we can use the singular value decomposition $W = UD_\lambda V^T$ of a rank-r matrix W to get

$$S_\lambda(\mathbf{W})\equiv\mathbf{U}\mathbf{D}_\lambda\mathbf{V}^T \quad \text{where} \quad \mathbf{D}_\lambda=\operatorname{diag}\left[(d_1-\lambda)_+,\ldots,(d_r-\lambda)_+\right].$$

The algorithm consists of these steps and is specified below:

1. Input: Matrix Z with missing values
2. Impute the missing values with some function (of observed values), e.g. column mean
3. Compute the SVD of the imputed matrix Z
4. Threshold the singular values. Replace missing values with corresponding imputed values obtained from low-rank approximation
5. Repeat steps 3 and 4 until convergence

Algorithm 7.1 SOFT-IMPUTE FOR MATRIX COMPLETION.

1. Initialize $\mathbf{Z}^{\text{old}}=\mathbf{0}$ and create a decreasing grid $\lambda_1>\ldots>\lambda_K$.
2. For each $k=1,\ldots,K$, set $\lambda=\lambda_k$ and iterate until convergence:
   Compute $\widehat{\mathbf{Z}}_\lambda\leftarrow\mathcal{S}_\lambda\big(P_\Omega(\mathbf{Z})+P_\Omega^\perp(\mathbf{Z}^{\text{old}})\big)$.
   Update $\mathbf{Z}^{\text{old}}\leftarrow\widehat{\mathbf{Z}}_\lambda$
3. Output the sequence of solutions $\widehat{\mathbf{Z}}_{\lambda_1},\ldots,\widehat{\mathbf{Z}}_{\lambda_K}$.

Source: Statistical Learning with Sparsity

Alternating Least Squares (ALS)

ALS is an iterative algorithm that uses matrix factorization where a small number of k is fixed and each user u is described by a k-dimensional vector x_k and each item i by a k-dimensional vector y_i. To predict a users rating we simply get $r_{ui}\approx x_u^T y_i$. Writing this as a matrices we have the $k\times n$ user matrix X and the $k\times m$ item matrix Y. The goal is to approximate $R\approx X^TY$. We can write this problem as

$$\min_{X,Y}\sum_{r_{ui}\,observed}(r_{ui}-x_u^{\intercal}y_i)^2+\lambda(\sum_u\|x_u\|^2+\sum_i\|y_i\|^2)$$

This objective is non-convex because of the $x_u^T y_i$ term and is in fact NP-hard to solve. To solve the problem, we alternatingly fix X and Y and treat them as constants which makes the objective convex. Therefore the Alternating Least Squares algorithm is defined as

Algorithm 1 ALS for Matrix Completion

Initialize X, Y

repeat

for $u = 1 \dots n$ **do**

$$x_u = (\sum_{r_{ui} \in r_{u*}} y_i y_i^\intercal + \lambda I_k)^{-1} \sum_{r_{ui} \in r_{u*}} r_{ui} y_i \tag{2}$$

end for

for $i = 1 \dots m$ **do**

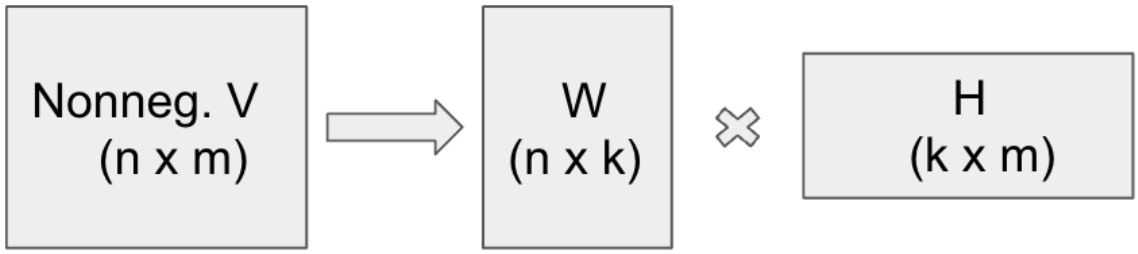
$$y_i = (\sum_{r_{ui} \in r_{*i}} x_u x_u^\intercal + \lambda I_k)^{-1} \sum_{r_{ui} \in r_{*i}} r_{ui} x_u \tag{3}$$

end for

until convergence

Source: https://stanford.edu/~rezab/classes/cme323/S15/notes/lec14.pdf

Nonnegative Matrix Factorization (NMF)



NMF approximates a matrix X with a low-rank matrix approximation such that $\mathbf{X} \approx \mathbf{WH}$.
In detail, we assume that X is set up. And there are n data points each with p dimensions, and every *column* of $\mathbf{X}$ is a data point, i.e. $\mathbf{X} \in \mathbb{R}^{p \times n}$.
For matrix W, each column is a basis element. According to these basis elements, we can reconstruct approximations to all of the original data points.
For matrix M, it interprets how to reconstruct an approximation to the original data point from a linear combination of the basis elements in W.
A popular method of measuring how good the approximation $\mathbf{WH}$ is using Frobenius norm. An optimal approximation to the Frobenius norm can be computed through truncated Singular Value Decomposition (SVD).
Basically, for implementing NMF, we need to consider the following optimization problem:

$$\min_{W \in \mathbb{R}^{p \times r}, H \in \mathbb{R}^{r \times n}} ||X - WH||_F^2 \quad \text{such that} \quad W \geq 0 \text{ and } H \geq 0.$$

The measuring method can be others. (e.g., Kullback-Leibler divergence for text-mining, the Itakura-Saito distance for music analysis, or the L1 norm to improve robustness against outliers.)
However, solving NMF has many issues: Firstly, solving the above optimization problem is a NP-hard problem. Another issue with NMF is that there is not guaranteed to be a single unique decomposition.
Fortunately there are heuristic approximations which have been proven to work well in many applications. There are several ways in which the **W** and **H** may be found. In this task, we use Lee and Seung's multiplicative update rule. This algorithm is: (type on LaTeX)

$$\text{initialize: } \mathbf{W} \text{ and } \mathbf{H} \text{ non negative. Then update the values in } \mathbf{W} \text{ and } \mathbf{H}$$

$$\text{by computing the following, with } n \text{ as an index of the iteration.}$$

$$\mathbf{H}_{[i,j]}^{n+1} \leftarrow \mathbf{H}_{[i,j]}^{n} \frac{\left((\mathbf{W}^n)^T \mathbf{V} \right)_{[i,j]}}{\left((\mathbf{W}^n)^T \mathbf{W}^n \mathbf{H}^n \right)_{[i,j]}}$$

and

$$\mathbf{W}_{[i,j]}^{n+1} \leftarrow \mathbf{W}_{[i,j]}^{n} \frac{\left(\mathbf{V} \left(\mathbf{H}^{n+1} \right)^T \right)_{[i,j]}}{\left(\mathbf{W}^n \mathbf{H}^{n+1} \left(\mathbf{H}^{n+1} \right)^T \right)_{[i,j]}}$$

$$\text{Until } \mathbf{W} \text{ and } \mathbf{H} \text{ are stable.}$$

Note that the updates are finished on a component by component premise not matrix multiplication.

Section 3 – Results and Comparison

KNN

Results for KNN on the MovieLens dataset. We also tried to tune the k to see the lowest RMSE (highlighted in dark red).

K	mean	median	zeros	min
3	0.4655	0.4846	0.9675	2.3939
5	0.4653	0.4844	0.9670	2.3943
10	0.4651	0.4843	0.9687	2.3959
20	0.4650	0.4842	0.9728	2.3984

50	0.4650	0.4841	0.9790	2.4041
100	0.4650	0.4841	0.9830	2.4072
500	0.4654	0.4842	0.9768	2.3994

Results for KNN on the Jester dataset. We also tried to tune the k to see the lowest RMSE (highlighted in dark red).

K	mean	median	zeros	min
3	0.8587	0.8598	0.9879	0.9891
5	0.8350	0.8361	0.9535	0.9548
10	0.8189	0.8204	0.9261	0.9269
20	0.8128	0.8148	0.9122	0.9140
50	0.8138	0.8163	0.9058	0.9076
100	0.8188	0.8216	0.9064	0.9082
500	0.8399	0.8431	0.9254	0.9278

Soft-Impute

Results for Soft-Impute on the MovieLens dataset using different fill methods. We also tried to subtract the column mean from each entry before imputing but this increases the RMSE drastically.

Fill method	mean	median	zeros	min
RMSE	0.47	0.48	1.25	1.23
RMSE (sub mean)	0.91	0.95	1.18	0.69

For the Jester dataset we applied the same methods:

Fill method	mean	median	zeros	min
RMSE	0.45	0.46	1.20	1.18
RMSE (sub mean)	0.81	0.85	1.18	0.68

Hard-Impute

We also applied Hard-Impute to both datasets. The first table shows the results for MovieLens and the second table shows the results for the Jester dataset.

Fill method	mean	median	zeros	min
RMSE	0.53	0.58	1.11	1.11
RMSE (sub mean)	0.91	0.95	1.10	0.66

Fill method	mean	median	zeros	min
RMSE	0.48	0.53	1.10	1.11
RMSE (sub mean)	0.84	0.85	1.11	0.63

Alternating Least Squares

For Alternating Least Squares we tried different parameters for *factors* and *regularization* but the RMSE is always around 2.5 for MovieLens and 2.9 for Jester so we discarded ALS.

Nonnegative Matrix Factorization

For the MovieLens dataset:

Fill method	mean	median	zeros	min
RMSE	1.326	1.40	2.38	2.11
MSE	1.76	0.157	13.39	7.53

Using 5-fold cross validation, we get the best components number is 16.

For the Jester dataset we applied the same methods:

Fill method	mean	median	zeros	min
RMSE	0.44	0.43	0.44	0.48

MSE	0.158	0.157	0.158	0.169

We get the best components number is 4.

Neural Matrix Factorization

In order to determine the best model using the column mean dataset, we experimented with different hyperparameters. Our observations indicate that the optimal parameters for the model are a learning rate of 0.001, a dropout rate of 0.5, embeddings of 16 and a multi-layer perceptron (MLP) hidden layer configuration of [64, 32, 16].

Tuning Hyper Parameter:
learning_rate choice [0.01,0.005,**0.001**,0.0005,0.0001]
dropout_prob choice [0.0,0.1,0.2,0.3,0.4,**0.5**]
mlp_hidden_size choice ['[**64,32,16**]',[32,16,8]']

The results obtained for the MovieLens dataset demonstrate the performance of various methods employed to address missing data using our best model.

Fill method	mean	median	zeros	min
RMSE	**0.448**	0.457	0.476	0.48
RMSE (sub mean)	0.996	1.06	1.12	1.23

This table represents the results for Jester dataset.

Fill method	mean	median	zeros	min
RMSE	**0.202**	0.583	1.32	1.21
RMSE (sub mean)	0.662	0.982	1.04	1.43

Comparison-

Algorithms (MovieLens Dataset)	**RMSE**
KNN	0.465
Nerual Matrix Factorization	**0.448**
Nonnegative Matrix Factorization	0.470
Alternating Least Squares	2.536

Hard-Impute	0.530
Soft-Impute	0.470

Algorithms (Jester Dataset)	**RMSE**
KNN	0.813
Nerual Matrix Factorization	**0.202**
Nonnegative Matrix Factorization	0.432
Alternating Least Squares	2.350
Hard-Impute	0.450
Soft-Impute	0.480

From The above comparison we observed that the NeuMF performed better and we used this algorithm to fill out Feedback.csv file.

Section 4 Difficulties:

KNN

The choice of k is crucial, because if k is small, it will generate a lot of noise to affect the imputation results. In our results, the lowest RMSE corresponds to a k value of no more than 100, and there are even cases where k=3. Therefore, it is necessary to further investigate how much noise interference there is.
Besides, KNN exists Curse of Dimensionality problem and is strongly influenced by outliers. However, our dataset is large in dimensionality, especially movielens, with more than 9000 columns, so we need to have further thinking about the performance of lower RMSE in movielens dataset.

Soft Imputation and Hard Imputation

Hard Imputation requires filling in missing values with reasonable replacements such as the mean or median. This can, however, cause a loss of information and lead to biases in the imputed MovieLens dataset. Additionally, assumptions about the data's distribution and properties may not be valid, resulting in inaccurate imputations.
Additionally, both soft imputation and hard imputation have the cold start problem. Because there is no historical data on which to base interpolation, it is difficult to interpolate for new users or new projects that do not have any scores

NeuMF and NMF

Finding the right hyperparameters for models like NeuMF and NMF was tricky. Especially with NeuMF, because it uses neural components, it took a lot more computing power and time to run experiments. Due to the limited computational power(mediocre laptop) we have, we took time constraint into consideration. We tried to achieve the best performance with minimum complexity of models. For instance, we could not put many layers in our NeuMF models because the training time increased exponentially as we put more layers.

Choosing the r value was a struggle with NMF too. To raise the r value, you needed a lot of computing power, which made it hard to explore the different settings within the time constraints.

We saw some big changes in the MSE values when we changed the hyperparameters or the amount of missing data. For example, NMF seemed to work really well when the missing percentage was between 10-20%, but if it went up to 50%, then it didn't work so well and other algorithms did better.

These issues with tuning hyperparameters and picking algorithms show how complicated it is to get the perfect recommendations for different collections of data and times when info is missing. It means that careful trials, time and effort have to be used to figure out which model works best in real life.

References

He, X., Liao, L., Zhang, H., Nie, L., Hu, X. and Chua, T.S., 2017, April. Neural collaborative filtering. In Proceedings of the 26th international conference on world wide web (pp. 173-182).

Gillis, Nicolas. "The Why and How of Nonnegative Matrix Factorization." Journal of Data Science, vol. 13, no. 2, 2015, pp. 381-388.

Lee, Daniel D., and H. Sebastian Seung. "Learning the parts of objects by non-negative matrix factorization." Nature, vol. 401, no. 6755, 1999, pp. 788-791.

Appendix: Important Parts of Code

Code for Nonnegative Matrix Factorization:

```python
# Generate a 5000x100 matrix with random values between 0 and 1
data_matrix = np.array(df)
# Replace some random entries with np.nan to represent missing values
missing_ratio = 0.1
mask = np.random.choice([True, False], data_matrix.shape, p=[missing_ratio, 1 - missing_ratio])
data_matrix_missing = data_matrix.copy()
data_matrix_missing[mask] = np.nan

# Define a custom scoring function for GridSearchCV
def nmf_mse_score(estimator, X):
    W = estimator.transform(X)
    H = estimator.components_
    completed_data = np.dot(W, H)
    completed_data_known = completed_data.copy()

    for i, j in zip(*np.where(~mask)):
        completed_data_known[i, j] = data_matrix[i, j]

    mse = mean_squared_error(data_matrix, completed_data_known)
    return -mse

# Search for the optimal number of components using GridSearchCV
param_grid = {"n_components": list([16,32,64,99])}
nmf = NMF()
grid_search = GridSearchCV(nmf, param_grid, scoring=nmf_mse_score, cv=5, n_jobs=-1, verbose=1)
grid_search.fit(np.nan_to_num(data_matrix_missing))

# Extract the best NMF model and number of components
best_nmf = grid_search.best_estimator_
best_n_components = grid_search.best_params_["n_components"]
print("Best number of components:", best_n_components)

# Reconstruct the completed matrix using the best NMF model
W = best_nmf.transform(np.nan_to_num(data_matrix_missing))
H = best_nmf.components_
completed_data = np.dot(W, H)

# Replace the known values with the original values in the completed matrix
completed_data_known = completed_data.copy()
for i, j in zip(*np.where(~mask)):
    completed_data_known[i, j] = data_matrix[i, j]
```

Code for NeuMF:

```
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split

data = np.array(df)

X_train, X_test = train_test_split(data, test_size=0.5, random_state=42)

num_users, num_items = data.shape
embedding_dim = 16

user_input = tf.keras.layers.Input(shape=(1,))
user_embedding = tf.keras.layers.Embedding(num_users, embedding_dim)(user_input)
user_vec = tf.keras.layers.Flatten()(user_embedding)
item_input = tf.keras.layers.Input(shape=(1,))
item_embedding = tf.keras.layers.Embedding(num_items, embedding_dim)(item_input)
item_vec = tf.keras.layers.Flatten()(item_embedding
gmf = tf.keras.layers.Dot(axes=1)([user_vec, item_vec])
concat = tf.keras.layers.Concatenate()([user_vec, item_vec])

dropout_rate = 0.2
mlp_layer = tf.keras.layers.Dense(64, activation='relu')(concat)
mlp_layer = tf.keras.layers.Dropout(dropout_rate)(mlp_layer)
mlp_layer = tf.keras.layers.Dense(32, activation='relu')(mlp_layer)
mlp_layer = tf.keras.layers.Dropout(dropout_rate)(mlp_layer)
mlp_layer = tf.keras.layers.Dense(32, activation='relu')(mlp_layer)
mlp_layer = tf.keras.layers.Dropout(dropout_rate)(mlp_layer)
mlp_layer = tf.keras.layers.Dense(32, activation='relu')(mlp_layer)
mlp_layer = tf.keras.layers.Dropout(dropout_rate)(mlp_layer)
neumf = tf.keras.layers.Concatenate()([gmf, mlp_layer])
output = tf.keras.layers.Dense(1, activation='linear')(neumf)
neural_mf = tf.keras.Model(inputs=[user_input, item_input], outputs=output)
neural_mf.compile(optimizer='adam', loss='mse')
neural_mf.fit([X_train[:, 0], X_train[:, 1]], X_train[:, 2], epochs=100, batch_size=32, validation_sp
```

Code for KNN:

```
# function calculating RMSE for tow dataframes without using sklaran.metrics
def rmse(df1, df2):
    return np.sqrt(((df1 - df2) ** 2).mean().mean())
```

```
df_nan = pd.read_csv('/content/mv_data_NaN.csv')
df_mean = df_nan.apply(lambda col: col.fillna(col.mean()), axis=0)
df_median = df_nan.apply(lambda col: col.fillna(col.median()), axis=0)
df_min = df_nan.apply(lambda col: col.fillna(col.min()), axis=0)
df_zero = df_nan.apply(lambda col: col.fillna(0), axis=0)
```

```
df_mask = np.array(df_mean)
# Replace some random entries with np.nan to represent missing values
missing_ratio = 0.5
mask = np.random.choice([True, False], df_mask.shape, p=[missing_ratio, 1 - missing_ratio])
df_mask50 = df_mask.copy()
df_mask50[mask] = np.nan
```

```
df_filled_knn_3_masked = KNN(k=3).fit_transform(df_mask50)
df_filled_knn_5_masked = KNN(k=5).fit_transform(df_mask50)
df_filled_knn_10_masked = KNN(k=10).fit_transform(df_mask50)
df_filled_knn_20_masked = KNN(k=20).fit_transform(df_mask50)
df_filled_knn_50_masked = KNN(k=50).fit_transform(df_mask50)
df_filled_knn_100_masked = KNN(k=100).fit_transform(df_mask50)
df_filled_knn_500_masked = KNN(k=500).fit_transform(df_mask50)
```