CLB-SVD V2.0

March 18, 2024

[1]:
```python
import pandas as pd
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import svds
from sklearn.metrics import mean_squared_error
from math import sqrt
```

[2]:
```python
#Pandas work with data in a table format [for data manipulation and analysis]
#NumPy is support for arrays and mathematical functions. [numerical computing]
# scipy.sparse.linalg.svds is a function to perform Singular Value␣
↪Decomposition (SVD) a way to decompose a matrix.
#sklearn.metrics.mean_squared_error calculates the mean squared error, a␣
↪measure of how close predictions are to the actual outcomes.
#sqrt from the math module calculates the square root.
```

[3]:
```python
# Parsing the data
def parse_data(file_path):
    data = []
    current_movie_id = None
    with open(file_path, 'r', encoding='utf-8-sig') as file:
        for line in file:
            try:
                if ':' in line:
                    current_movie_id = int(line.split(':')[0])
                else:
                    customer_id, rating, _ = line.strip().split(',')
                    data.append([current_movie_id, int(customer_id),␣
↪int(rating)])
            except ValueError:
                # This handles lines that don't have the expected format
                continue  # Skips to the next line
    return pd.DataFrame(data, columns=['MovieID', 'CustomerID', 'Rating'])

# Function to load data for training, test, and validation
def load_dataset(file_paths):
    data_frames = [parse_data(file_path) for file_path in file_paths]␣
↪#parse_data function to load and structure the data into a DataFrame
```

```python
    combined_data = pd.concat(data_frames)
    return combined_data

# File paths setup
train_files = ['training_set_c1.txt','training_set_c2.txt']
test_file = ['test_set_c1.txt', 'test_set_c2.txt']
validation_file = ['validation_set_c1.txt','validation_set_c2.txt']

# Load datasets
train_movie_data = load_dataset(train_files)

train_movie_data.head()
```

[3]:

	MovieID	CustomerID	Rating
0	1	401047	4
1	1	14756	4
2	1	2566259	5
3	1	1398626	2
4	1	1294335	2

[4]:
```python
# Converts the DataFrame into a user-item matrix where rows represent␣
↪customers, columns represent movies, and values are ratings.
# Missing ratings are filled with 0, indicating unrated movies.
```

[5]:
```python
# Create Training user-item matrix
train_ratings_df = train_movie_data.pivot(index='CustomerID',␣
↪columns='MovieID', values='Rating').fillna(0)

# Convert the DataFrame to a Compressed Sparse Column (CSC) matrix
train_ratings_matrix  = csc_matrix(train_ratings_df.values)

num_users, num_movies = train_ratings_matrix .shape
print(f"Number of users: {num_users}, Number of movies: {num_movies}")
```

```
/tmp/ipykernel_15519/1906747628.py:2: PerformanceWarning: The following
operation may generate 4384890210 cells in the resulting pandas object.
  train_ratings_df = train_movie_data.pivot(index='CustomerID',
columns='MovieID', values='Rating').fillna(0)

Number of users: 476101, Number of movies: 9210
```

[6]:
```python
#Performs SVD on the sparse matrix, decomposing it into matrices U, Sigma, and␣
↪V^T. Sigma is converted into a diagonal matrix.
```

[7]:
```python
# "k" value represents the number of singular values (or latent factors)
k=11
# Perform SVD with the updated k value
```

```
U, sigma, Vt = svds(train_ratings_matrix, k=k)
sigma = np.diag(sigma)
```

[8]:
```
## SVD is a technique used to break down the ratings matrix into three smaller␣
↪matrices.
      #It helps to identify patterns in the data, such as similar preferences␣
↪among customers.
## k is the number of latent factors you want to keep. These factors are like␣
↪hidden themes that capture the essence of the data.
```

[9]:
```
def predict(matrix, U, sigma, Vt):
    # Calculate mean user rating with correct reshaping
    mean_user_rating = matrix.mean(axis=1).reshape(-1, 1)
    # Ensure that U, sigma, and Vt are correctly aligned with the input matrix␣
↪dimensions
    preds = np.dot(np.dot(U, sigma), Vt) + mean_user_rating
    return preds
```

[10]:
```
#This line reconstructs the ratings matrix using the decomposed matrices␣
↪obtained from SVD, adjusted by the average rating for each user.
#It essentially predicts the ratings that users might give to movies they␣
↪haven't rated.
```

[11]:
```
# RMSE calculation
def calculate_rmse(actual, predicted):
    # Ensure both actual and predicted are numpy arrays
    mask = actual.nonzero()
    actual_filtered = actual[mask].flatten()
    predicted_filtered = predicted[mask].flatten()
    return sqrt(mean_squared_error(actual_filtered, predicted_filtered))

# Assuming train_ratings_matrix is your actual ratings and train_preds is your␣
↪predictions
train_preds = predict(train_ratings_df.values, U, sigma, Vt)
# Calculate RMSE
print('Training RMSE:', calculate_rmse(train_ratings_matrix.toarray(),␣
↪train_preds))
```

```
Training RMSE: 2.8289205193008806
```

[12]:
```
#RMSE (Root Mean Square Error) is a way to measure how accurate the predictions␣
↪are. It calculates the difference between the predicted ratings and the␣
↪actual ratings given by the users, providing a single number that represents␣
↪the average error.
#A lower RMSE means the predictions are closer to the actual ratings,␣
↪indicating a better performing model.
```

[13]:
```python
test_movie_data = parse_data(test_file[0])
```

[14]:
```python
# Create a Test user-item matrix
test_ratings_df= test_movie_data.pivot(index='CustomerID', columns='MovieID',␣
↪values='Rating').reindex(index=train_ratings_df.index,␣
↪columns=train_ratings_df.columns).fillna(0)

# Convert to CSC format
test_ratings_matrix = csc_matrix(test_ratings_df.values)

num_users, num_movies = test_ratings_matrix.shape
print(f"Number of users: {num_users}, Number of movies: {num_movies}")
```

```
Number of users: 476101, Number of movies: 9210
```

[15]:
```python
# Make predictions with adjusted dimensions
test_preds = predict(test_ratings_df.values, U, sigma, Vt)
```

[16]:
```python
print("Shape of test_matrix.values:", test_ratings_df.values.shape)
print("Shape of test_preds:", test_preds.shape)
```

```
Shape of test_matrix.values: (476101, 9210)
Shape of test_preds: (476101, 9210)
```

[17]:
```python
def calculate_rmse(actual, predicted):
    mask = actual > 0  # Assuming ratings are positive and 0 indicates missing␣
↪rating
    print(f"Non-zero entries in 'actual': {np.sum(mask)}")

    actual_filtered = actual[mask]
    predicted_filtered = predicted[mask]
    print(f"Shape of 'actual': {actual.shape}")
    print(f"Shape of 'predicted': {predicted.shape}")

    return sqrt(mean_squared_error(actual_filtered, predicted_filtered))

# Calculate RMSE

print('Test RMSE:', calculate_rmse(test_ratings_matrix.toarray(), test_preds))
```

```
Non-zero entries in 'actual': 3607424
Shape of 'actual': (476101, 9210)
Shape of 'predicted': (476101, 9210)
Test RMSE: 2.948704925212078
```

[18]:
```python
validation_movie_data = parse_data(validation_file[0])
```

[19]:

```
# Create a validation matrix similar to the training matrix
validation_ratings_df = validation_movie_data.pivot(index='CustomerID',␣
↪columns='MovieID', values='Rating').reindex(index=train_ratings_df.index,␣
↪columns=train_ratings_df.columns).fillna(0)
# Convert to CSC format
validation_ratings_matrix = csc_matrix(validation_ratings_df.values)

num_users, num_movies = validation_ratings_matrix.shape
print(f"Number of users: {num_users}, Number of movies: {num_movies}")
```

```
Number of users: 476101, Number of movies: 9210
```

[20]:

```
# Make predictions with adjusted dimensions
validation_preds = predict(validation_ratings_df.values, U, sigma, Vt)
```

[21]:

```
print("Shape of validation_matrix.values:", validation_ratings_df.values.shape)
print("Shape of validation_preds:", validation_preds.shape)
```

```
Shape of validation_matrix.values: (476101, 9210)
Shape of validation_preds: (476101, 9210)
```

[22]:

```
print('validation RMSE:', calculate_rmse(validation_ratings_matrix.toarray(),␣
↪validation_preds))
```

```
Non-zero entries in 'actual': 3607427
Shape of 'actual': (476101, 9210)
Shape of 'predicted': (476101, 9210)
validation RMSE: 2.9491639886583707
```

[ ]:

[ ]:

[ ]: