Collaborative Filtering: Matrix Factorization

This document offers a comprehensive explanation of a matrix factorization-based collaborative filtering's implementation in Python. The primary objective is to develop a recommendation model that leverages user ratings for items (such as movies) to suggest similar items to users. The implementation focuses on optimizing the underlying model through iterative training, aiming to deliver personalized recommendations by capturing user-item interactions effectively.

Code walkthrough:

The code implement matrix factorization from scratch-

Creating the Ratings Matrix

```
[60] new_ratings = csr_matrix(
        (ratings.rating.values, (ratings.user.values, ratings.movie.values)),
        shape=(len(ratings.user.unique()), len(ratings.movie.unique()))
    ).toarray()
```

- `csr_matrix`: Creates a sparse matrix to efficiently represent the ratings matrix.
- The `csr_matrix` constructor takes three arguments:
  - `ratings.rating.values`: The actual rating values.
  - `(ratings.user.values, ratings.movie.values)`: The row and column indices corresponding to users and movies.
  - `shape`: Specifies the dimensions of the matrix based on the number of unique users and movies.
- `.toarray()`: Converts the sparse matrix into a NumPy array for ease of manipulation.

Defining the Matrix Factorization Class

```python
class MatrixFactorization():

    def __init__(self, ratings, n_factors=100, l_rate=0.01, alpha=0.01, n_iter=100):
        self.ratings = ratings
        self.n_users, self.n_items = ratings.shape
        self.non_zero_row_ind, self.non_zero_col_ind = ratings.nonzero()
        self.n_interac = len(ratings[np.where(ratings != 0)])
        self.ind_lst = list(range(self.n_interac))
        self.n_factors = n_factors
        self.l_rate = l_rate  # eta0 Constant that multiplies the update term
        self.alpha = alpha  # lambda Constant that multiplies the regularization term
        self.n_iter = n_iter
        self.mse_lst = []
        self.wait = 10
        self.tol = 1e-3
        self.n_iter_no_change = 10
        self.verbose = True
        self.stop = False
```

Key components:

- `self.ratings`: Stores the user-item ratings matrix.
- `self.n_users`, `self.n_items`: Determine the number of users and items in the matrix.
- `self.non_zero_row_ind`, `self.non_zero_col_ind`: Store the indices of non-zero entries (user-item pairs with ratings).
- `self.n_interac`: Total number of non-zero interactions in the matrix.
- `self.n_factors`: The number of latent features (embedding dimensions).
- `self.l_rate`: Learning rate for SGD updates.
- `self.alpha`: Regularization strength to prevent overfitting.
- `self.n_iter`: Maximum training epochs.
- Early stopping parameters (`self.wait`, `self.tol`, etc.) are initialized to monitor model performance.

Initialization of Biases and Embeddings

```python
def initialize(self, ):
    self.now = time.time()
    # Initialize Bias Values
    self.user_biases = np.zeros(self.n_users)
    self.item_biases = np.zeros(self.n_items)
    # initialize user & item vectors
    self.user_vecs = np.random.normal(scale=1/self.n_factors, size=(self.n_users, self.n_factors))
    self.item_vecs = np.random.normal(scale=1/self.n_factors, size=(self.n_items, self.n_factors))
    # compute global bias
    self.global_bias = np.mean(self.ratings[np.where(self.ratings != 0)])
    self.evaluate_the_model(0)
```

Purpose: Initializes model parameters.

- `self.user_biases` and `self.item_biases`: Initialized as zeros to represent no bias initially.
- `self.user_vecs` and `self.item_vecs`: Initialized with small random values.
- `self.global_bias`: Represents the average rating across all interactions

Prediction

```python
def predict(self, u, i):
    return self.global_bias+self.user_biases[u]+self.item_biases[i]+self.user_vecs[u]@self.item_vecs[i]
```

Computes the predicted rating for user `u` and item `i` by combining:

- The global bias.
- User-specific and item-specific biases.
- Dot product of user and item embeddings.

Updating Biases and Embeddings

```
def update_biases_and_vectors(self, error, u, i):
    # Update biases
    self.user_biases[u] += self.l_rate*(error - self.alpha*self.user_biases[u])
    self.item_biases[i] += self.l_rate*(error - self.alpha*self.item_biases[i])
    # Update User and item Vectors
    self.user_vecs[u, :] += self.l_rate*(error*self.item_vecs[i, :] - self.alpha*self.user_vecs[u, :])
    self.item_vecs[i, :] += self.l_rate*(error*self.user_vecs[u, :] - self.alpha*self.item_vecs[i, :])
```

- Updates biases (`user_biases` and `item_biases`) and embeddings (`user_vecs` and `item_vecs`) using the error between the true and predicted rating.
- Regularization (`self.alpha`) is applied to prevent overfitting by penalizing large bias or embedding values.

Model evaluation

```
def evaluate_the_model(self, epoch):
    tot_square_error = 0
    for index in self.ind_lst:
        # Extracting user item information indices in which we have a rating
        u, i = self.non_zero_row_ind[index], self.non_zero_col_ind[index]
        pred_rat = self.predict(u, i)
        tot_square_error += (self.ratings[u, i]-pred_rat)**2
    mse = tot_square_error/self.n_interac
    self.mse_lst.append(mse)
    if self.verbose:
        print(f"---> Epoch {epoch}")
        temp = np.round(time.time()-self.now, 3)
        print(f"ave mse {np.round(self.mse_lst[-1], 3)} ===> Total training time: {temp} seconds.")
```

- Loops through all interactions (non-zero entries in the ratings matrix).
- Extracts the user (`u`) and item (`i`) indices from `self.ind_lst`.
- Predicts the rating for the user-item pair using `self.predict`.
- Calculates the squared error between the predicted and actual rating and accumulates it.
- Divides the total squared error by the number of interactions to get the MSE.

- Prints the training progress.

Checking the convergence

```
def early_stopping(self, epoch):
    if (self.mse_lst[-2] - self.mse_lst[-1]) <= self.tol:
        if self.wait == self.n_iter_no_change:
            temp = np.round(time.time()-self.now, 3)
            if self.verbose: print(f"Convergence after {epoch} epochs time took: {temp} seconds.")
            self.stop = True
            self.conv_epoch_num = epoch
        self.wait += 1
    else:
        self.wait = 0
```

- Compares the MSE improvement between the last two epochs to the predefined tolerance (`self.tol`).
- If the improvement is below the tolerance for `self.n_iter_no_change` consecutive epochs, training stops.
- Resets the patience counter (`self.wait`) if the improvement is significant.

Training the model

```
def fit(self, ):
    self.initialize()
    for epoch in range(1, self.n_iter):
        np.random.shuffle(self.ind_lst)
        if self.stop == False:
            for index in self.ind_lst:
                # Extracting user item information indices in which we have a rating
                u, i = self.non_zero_row_ind[index], self.non_zero_col_ind[index]
                pred_rat = self.predict(u, i)
                error = self.ratings[u, i]-pred_rat
                self.update_biases_and_vectors(error, u, i)
            self.evaluate_the_model(epoch)
            self.early_stopping(epoch)
    self.plot_the_score()
```

- Initializes the model parameters (biases, embeddings, etc.).

- Repeats the training process for a fixed number of epochs (`self.n_iter`) or until convergence.
- In each epoch:
    - The order of training data is shuffled.
    - Predictions and updates are made for each user-item interaction.
    - Early stopping monitors the progress and stops training if convergence is detected.