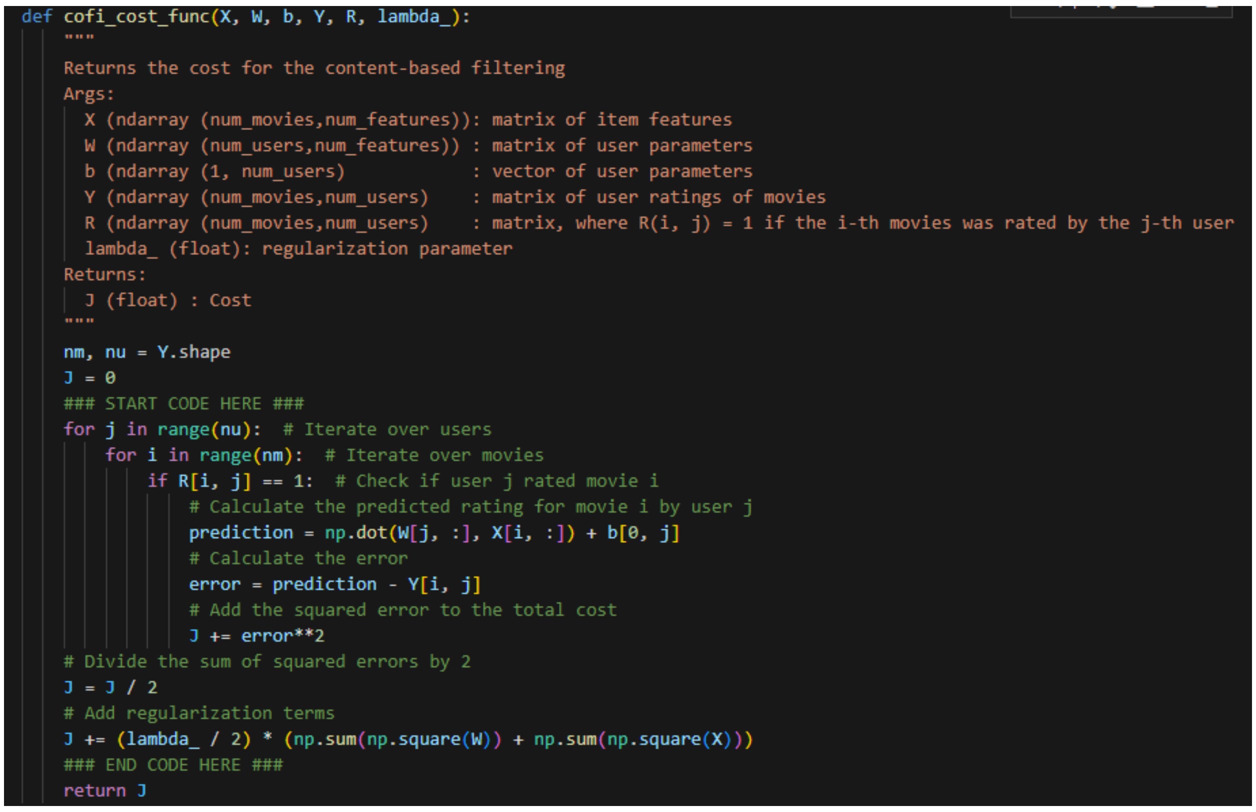
Algorithm



```
def cofi_cost_func(X, W, b, Y, R, lambda_):
    """
    Returns the cost for the content-based filtering
    Args:
      X (ndarray (num_movies,num_features)): matrix of item features
      W (ndarray (num_users,num_features)) : matrix of user parameters
      b (ndarray (1, num_users)            : vector of user parameters
      Y (ndarray (num_movies,num_users)    : matrix of user ratings of movies
      R (ndarray (num_movies,num_users)    : matrix, where R(i, j) = 1 if the i-th movies was rated by the j-th user
      lambda_ (float): regularization parameter
    Returns:
      J (float) : Cost
    """
    nm, nu = Y.shape
    J = 0
    ### START CODE HERE ###
    for j in range(nu):  # Iterate over users
        for i in range(nm):  # Iterate over movies
            if R[i, j] == 1:  # Check if user j rated movie i
                # Calculate the predicted rating for movie i by user j
                prediction = np.dot(W[j, :], X[i, :]) + b[0, j]
                # Calculate the error
                error = prediction - Y[i, j]
                # Add the squared error to the total cost
                J += error**2
    # Divide the sum of squared errors by 2
    J = J / 2
    # Add regularization terms
    J += (lambda_ / 2) * (np.sum(np.square(W)) + np.sum(np.square(X)))
    ### END CODE HERE ###
    return J
```

Okay, let's break down the cofi_cost_func function step by step. This function is a cornerstone in collaborative filtering, as it calculates the "cost" or "error" of your recommender system's current predictions. The lower the cost, the better your system is at predicting movie ratings.

Overall Purpose:

The function cofi_cost_func calculates the cost for a collaborative filtering model. This cost has two main components:

1. **Squared Error Loss:** This measures how different the model's predicted ratings are from the actual ratings given by users. It only considers movies that users have actually rated.
2. **Regularization:** This term helps prevent overfitting by penalizing large values in the learned movie features (X) and user parameters (W). Overfitting is when the model learns the training data too well, including its noise, and performs poorly on new, unseen data.

Let's look at the code line by line:

unfold_lesspython

content_copyadd

# GRADED FUNCTION: cofi_cost_func

# UNQ_C1

def cofi_cost_func(X, W, b, Y, R, lambda_):

  """

  Returns the cost for the content-based filtering

  Args:

    X (ndarray (num_movies,num_features)): matrix of item features

W (ndarray (num_users,num_features)) : matrix of user parameters

b (ndarray (1, num_users)   : vector of user parameters

Y (ndarray (num_movies,num_users)   : matrix of user ratings of movies

R (ndarray (num_movies,num_users)   : matrix, where R(i, j) = 1 if the i-th movies was rated by the j-th user

lambda_ (float): regularization parameter

Returns:

J (float) : Cost

"""

- **Function Definition:**
  - def cofi_cost_func(X, W, b, Y, R, lambda_): defines the function with its parameters.
- **Arguments (Inputs):**
  - X: A NumPy array where each row X[i, :] is a vector of features for movie i. The shape is (num_movies, num_features).
  - W: A NumPy array where each row W[j, :] is a parameter vector for user j. These parameters capture the user's tastes. The shape is (num_users, num_features).
  - b: A NumPy array (a row vector) where b[0, j] is a bias term for user j. Some users might generally give higher or lower ratings, and the bias accounts for this. The shape is (1, num_users).
  - Y: A NumPy array containing the actual ratings. Y[i, j] is the rating user j gave to movie i. If a movie isn't rated, this value might be 0 (though the R matrix is the definitive source for whether a rating exists). The shape is (num_movies, num_users).
  - R: A binary NumPy array (contains only 0s and 1s). R[i, j] = 1 if user j rated movie i, and R[i, j] = 0 otherwise. This is crucial because we only want to calculate the error for movies that have been rated.
  - lambda_: A floating-point number that controls the strength of the regularization. A higher lambda_ means a stronger penalty for large W and X values.

nm, nu = Y.shape

J = 0

- nm, nu = Y.shape: This unpacks the dimensions of the Y matrix. nm becomes the number of movies, and nu becomes the number of users.
- J = 0: Initializes the cost J to zero. We will accumulate the cost in this variable.

### START CODE HERE ###

```
 for j in range(nu):  # Iterate over users
   for i in range(nm):  # Iterate over movies
     if R[i, j] == 1:  # Check if user j rated movie i
```

- **Nested Loops:**
  - for j in range(nu):: The outer loop iterates through each user, from user 0 to nu-1.
  - for i in range(nm):: The inner loop iterates through each movie, from movie 0 to nm-1.
- **Conditional Check:**
  - if R[i, j] == 1:: This is a very important condition. It checks the R matrix to see if the current user j has actually rated the current movie i. If R[i, j] is 0, it means there's no rating, so we skip the error calculation for this movie-user pair.

```
       # Calculate the predicted rating for movie i by user j
       prediction = np.dot(W[j, :], X[i, :]) + b[0, j]
```

- **Prediction Calculation (if rated):**
  - This line calculates the model's predicted rating for movie i by user j.
  - W[j, :]: Selects the entire row for user j from the W matrix (the parameter vector for user j).
  - X[i, :]: Selects the entire row for movie i from the X matrix (the feature vector for movie i).
  - np.dot(W[j, :], X[i, :]): Computes the dot product of the user's parameter vector and the movie's feature vector. This is the core of the collaborative filtering prediction. It essentially measures how well the user's preferences (W[j,:]) align with the movie's characteristics (X[i,:]).
  - + b[0, j]: Adds the bias term for user j.

```
       # Calculate the error
       error = prediction - Y[i, j]
```

- **Error Calculation:**
  - error = prediction - Y[i, j]: Calculates the difference between the prediction we just made and the actual rating Y[i, j] that user j gave to movie i.

```
       # Add the squared error to the total cost
       J += error**2
```

- **Accumulating Squared Error:**
  - J += error**2: The error is squared (to make all errors positive and to penalize larger errors more) and then added to the total cost J.

```
 # Divide the sum of squared errors by 2
 J = J / 2
```

- **Averaging Factor (1/2):**

- After the loops have finished summing up all the squared errors for the rated items, J is divided by 2. This 1/2 factor is a mathematical convention often used in cost functions involving squared errors. It simplifies the derivative of the cost function when it's used in optimization algorithms like gradient descent (the '2' from the power rule of differentiation cancels out).

```
# Add regularization terms
J += (lambda_ / 2) * (np.sum(np.square(W)) + np.sum(np.square(X)))
```

- **Regularization Term:**
    - This line adds the regularization penalty to the cost J.
    - np.square(W): Computes the element-wise square of all values in the W matrix.
    - np.sum(np.square(W)): Sums up all these squared values in W. This is $\sum_{j=0}^{n_u-1}\sum_{k=0}^{n-1}(W_{jk})^2$.
    - np.square(X): Computes the element-wise square of all values in the X matrix.
    - np.sum(np.square(X)): Sums up all these squared values in X. This is $\sum_{i=0}^{n_m-1}\sum_{k=0}^{n-1}(X_{ik})^2$.
    - (np.sum(np.square(W)) + np.sum(np.square(X))): The sum of the squared elements of both W and X.
    - (lambda_ / 2) * ...: This sum is then multiplied by lambda_ / 2.
        - lambda_: The regularization parameter. If lambda_ is 0, there's no regularization. If lambda_ is large, the model is heavily penalized for having large W or X values, encouraging simpler models.
        - The / 2 here is also for mathematical convenience when calculating derivatives.

```
### END CODE HERE ###


return J
```

- **Return Cost:**
    - return J: The function returns the final calculated cost J. This value represents how well (or poorly) the current parameters X, W, and b fit the observed ratings Y, considering the regularization penalty.

In summary, the cofi_cost_func implements the following mathematical formula:

$J = \frac{1}{2} \sum_{(i,j):R(i,j)=1} (\mathbf{w}^{(j)} \cdot \mathbf{x}^{(i)} + b^{(j)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=0}^{n_u-1} \sum_{k=0}^{n-1} (w_k^{(j)})^2 + \frac{\lambda}{2} \sum_{i=0}^{n_m-1} \sum_{k=0}^{n-1} (x_k^{(i)})^2$

The first part is the sum of squared errors for all rated movie-user pairs. The second and third parts are the regularization terms for the user parameters W and movie features X, respectively. The goal of the learning algorithm will be to find X, W, and b that minimize this J.