Recommender systems implementation details

Mean normalization

In recommender systems or any model that relies on ranking values, **scaling the features** is **crucial** because t**he raw values** alone **lack meaningful context.**

Example: With our model so far, if a new user register to our website and has not rated any movies yet,
our model will predict that they will rate zero to all movies. $(?) \rightarrow$
$0\ (\text{what we don't want})$

A common approach for this is **mean normalization**, which ensures that features are on a **consistent scale.** This technique is also particularly helpful when users have missing or undefined values in the dataset. By computing the average rating for each item, the system can still generate recommendations—even for new users with no prior data—by using these averages as default estimates. Essentially, the item mean acts as the model's initial prediction before it begins adjusting through parameters like w and b.

To perform *mean normalization*, the average value of each item across all users is calculated and stored in a vector. Suppose n is the number of items and m is the number of users; these averages provide the baseline for scaling and recommendation.

$$\text{TABLE DATA} \rightarrow \text{Matrix } Y = \begin{bmatrix} y^{(0,0)} & \dots & y^{(0,m)} \\ \vdots & \ddots & \vdots \\ y^{(n,0)} & \dots & y^{(n,m)} \end{bmatrix} \text{ and } \mu = \begin{bmatrix} \text{mean}(y^0) \\ \vdots \\ \text{mean}(y^n) \end{bmatrix}$$

From Y and μ, we calculate new $Y_{normalized}$:

$$Y = \begin{bmatrix} y^{(0,0)} - \mu_0 & \dots & y^{(0,m)} - \mu_0 \\ \vdots & \ddots & \vdots \\ y^{(n,0)} - \mu_n & \dots & y^{(n,m)} - \mu_n \end{bmatrix}$$

Since this process generates negative data points, the model function needs to account for them. Otherwise, it may output negative ratings that would later need to be re-normalized. To avoid this extra step, the mean normalization vector can be incorporated directly into the model function. Then, for user j, on the movie i predict:

$$f_{w,b}(x) = w^{(j)} \cdot x^{(i)} + b^{(j)} + \mu_i$$

$\Rightarrow$ This resolves the issue since each new user will now be given the average movie ratings from existing users.

Example:j

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}; \quad \mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix} \Rightarrow Y' \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

User 5 (Eve):

$$w^{(5)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad b^{(5)} = 0$$
$$\Rightarrow \underbrace{w^{(5)} \cdot x^{(1)} + b^{(5)}}_{0} + \mu_1 = 2.5$$

TensorFlow implementation of collaborative filtering

TensorFlow provides a built-in module, `tf.GradientTape`, which enables automatic differentiation in just a single line. It also keeps track of the operations used to compute the cost function, allowing them to be reused later. (usually call `Auto-Diff` or `Auto-Grad`).

Custom training loop

`tF.variables` are the parameters that we want to optimize. `tF.variables` require special function (`assign_add()`) to modify.

```
w = tf.Variable(3.0)
x = 1.0
y = 1.0 # target value
alpha = 0.01

iterations = 30
for iter in range(iterations):
    # Use TensorFlow's Gradient tape to record the steps
    # used to compute the cost J, to enable auto differentiation.
    with tf.GradientTape() as tape:
        f_wb = w*x
          cost = (f_wb - y)**2

    # Use the gradient tape to calculate the gradients of
    # the cost wiht respect to the parameter w.
    [dJdw] = tape.gradient(cost, [w])

    # Run one step of gradient descent by updating
    # the value of w to reduce the cost.
    w.assign_add(-alpha * dJdw)
```

Implementation in TensorFlow

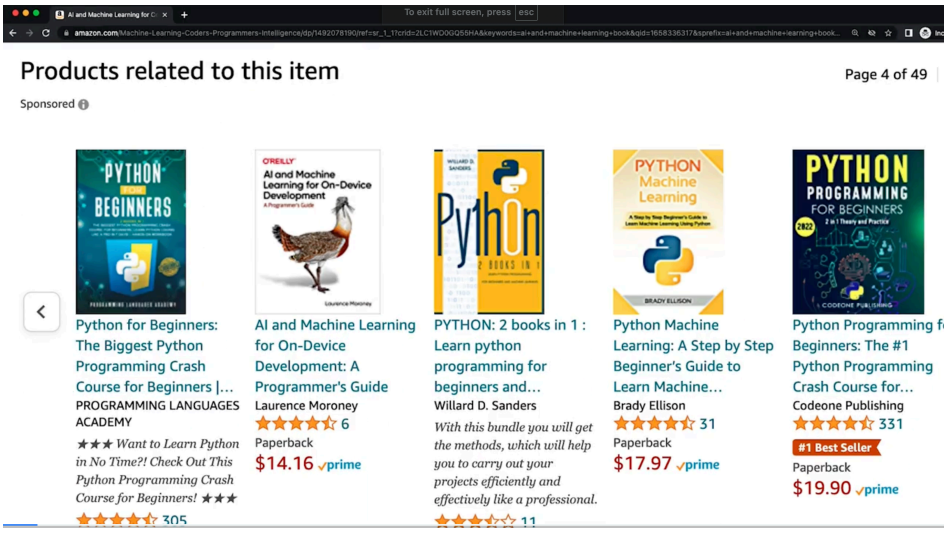
- `Y_norm` is the set of target values normalized.
- `R` defines which datapoints have ratings (so **null** values are excluded)

```
# Instantiate an optimizer
optimizer  = keras.optimizers.Adam(learning_rate=1e-1)

iterations = 200
for iter in range(iterations):
    # User TensorFlow's GradientTape to creord the
    # operations used to compute the cost
    with tf.GradientTape() as tape:

        # Compute the cost (forward pass is included in cost)
        cost_value = cofiCostFuncV(X, W, b, Ynorm, R, n, m, lambda)

    # Use the gradient tape to automatically retrieve the
    # gradients fo the trainable variables with respect to loss
    grads = tape.gradient(cost_value, [X,W,b])

    # Run one step of the gradient descent by updating the
    # value of the variables to minimize the loss
    optimizer.apply_gradients(zip(grads, [X,W,b]))
```

`cofiCostFuncV()` is similar to: repeat $\left[\begin{array}{l} w_i^{(j)} = w_i^{(j)} - \alpha \dfrac{\partial}{\partial w_i^{(j)}} J(w,b,x) \\ b^{(j)} = b^{(j)} - \alpha \dfrac{\partial}{\partial b^{(j)}} J(w,b,x) \\ x_k^{(i)} = x_k^{(i)} - \alpha \dfrac{\partial}{\partial x_k^{(i)}} J(w,b,x) \end{array}\right]$

Finding related items

- The algorithm identifies similar items by analyzing features of items, such as genre for books.
- It calculates the squared distance between feature vectors of items to find those that are most similar.



Collaborative filtering learns **feature vectors** $x^{(i)}$ for every item (movie, book, product, etc.), which represent the characteristics of that item.

- Unlike manually labeled features (like "romance" or "action"), these learned features $x^{(i)}$ are often **hard to interpret individually**, but collectively they capture meaningful patterns about an item.
- To find items related to a specific item iii, you look for other items kkk whose feature vectors $x^{(k)}$ are **close to** $x^{(i)}$.
- The closeness or similarity between feature vectors is measured by the **squared distance**:

$$\sum_{l=1}^{n} (x_l^{(k)} - x_l^{(i)})^2 = ||x_l^{(k)} - x_l^{(i)}||^2$$

- Items with feature vectors that minimize this distance are considered the most similar.
- By finding the top 5, 10, or more similar items, websites provide related recommendations such as **"more movies like this"** or **"books similar to this one."**

Limitations of Collaborative Filtering

1. **Cold Start Problem:**
    - Collaborative filtering requires **user ratings.**
    - New **items** with **few** or **no** ratings are hard to recommend accurately.
    - New **users** with very **few** ratings make it hard to **personalize recommendations**.
    - Techniques like **mean normalization** help, but the cold start remains a known challenge.
2. **Limited Use of Side Information:**
    - Collaborative filtering does **not naturally incorporate additional info** about items or users.
    - For example, movie genre, cast, studio, budget, or user demographics (age, gender, location) aren't directly used.
    - Even subtle cues like web browser or device type can correlate with user preferences but are not integrated in collaborative filtering.
3. **Grey sheep problem**

Because of these limitations, **content-based filtering** methods, which use item/user attributes explicitly, are often combined with collaborative filtering to improve recommendation quality.

$\Rightarrow$ Content-based filtering can address cold start and leverage additional data.

Lecture described using ‘mean normalization’ to do feature scaling of the ratings. What equation below best describes this algorithm?

○

$$y_{norm}(i,j) = \frac{y(i,j) - \mu_i}{\sigma_i} \quad \text{where}$$
$$\mu_i = \frac{1}{\sum_j r(i,j)} \sum_{j:r(i,j)=1} y(i,j)$$
$$\sigma_i^2 = \frac{1}{\sum_j r(i,j)} \sum_{j:r(i,j)=1} (y(i,j) - \mu_j)^2$$

○

$$y_{norm}(i,j) = \frac{y(i,j) - \mu_i}{max_i - min_i} \quad \text{where}$$
$$\mu_i = \frac{1}{\sum_j r(i,j)} \sum_{j:r(i,j)=1} y(i,j)$$

◉

$$y_{norm}(i,j) = y(i,j) - \mu_i \quad \text{where}$$
$$\mu_i = \frac{1}{\sum_j r(i,j)} \sum_{j:r(i,j)=1} y(i,j)$$

Explain: This is the mean normalization algorithm described in lecture. This will result in a zero average value on a per-row basis.

The implementation of collaborative filtering utilized a custom training loop in TensorFlow. Is it true that TensorFlow always requires a custom training loop?

- No: TensorFlow provides simplified training operations for some applications. 👍👍
- Yes. TensorFlow gains flexibility by providing the user primitive operations they can combine in many ways.

Explain: Recall in Course 2, you were able to build a neural network using a ‘model’, ‘compile’, ‘fit’, sequence which managed the training for you. A custom training loop was utilized in this situation because training w, b and x does not fit the standard layer paradigm of TensorFlow's neural network flow. There are alternate solutions such as custom layers, however, it is useful in this course to introduce you to this powerful feature of TensorFlow.

Once a model is trained, the 'distance' between features vectors gives an indication of how similar items are.

The squared distance between the two vectors $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(i)}$ is:

$$distance = \left\|\mathbf{x^{(k)}} - \mathbf{x^{(i)}}\right\|^2 = \sum_{l=1}^{n}(x_l^{(k)} - x_l^{(i)})^2$$

Using the table below, find the closest item to the movie "Pies, Pies, Pies".

Movie	User 1	…	User n	x_0	x_1	x_2
Pastries for Supper				2.0	2.0	1.0
Pies, Pies, Pies				2.0	3.0	4.0
Pies and You				5.0	3.0	4.0

Answer: Pies and You: $distance = 9$

Which of these is an example of the cold start problem? (Check all that apply.)

☐ A recommendation system takes so long to train that users get bored and leave.

☐ A recommendation system is so computationally expensive that it causes your computer CPU to heat up, causing your computer to need to be cooled down and restarted.

☑ ~~A recommendation system is unable to give accurate rating predictions for a new user that has rated few products.~~

☑ ~~A recommendation system is unable to give accurate rating predictions for a new product that no users have rated.~~

Explain:

- A recommendation system uses user feedback to fit the prediction model.
- A recommendation system uses product feedback to fit the prediction model.