Introduction

Recommendation systems try to recommend items (products, music, scientific papers, movies, services, etc) to interested potential customers, based on the information available. A successful recommendation system can significantly improve the revenue of e-commerce or facilitate the interaction of users in online communities.

There are many methods for recommender systems. Among them we consider some of best methods for our dataset.

In Collaborative Filtering, we're going to find users which are similar to me. This technique aims to fill the missing entries of a user-item association matrix.

1. **Collaborative Filtering**-Formula for prediction is based on probability formulas
   User score (user-user collaborating filtering):

$$s(i,j) = \bar{r}_i + \frac{\sum_{i' \in \Omega_j} w_{ii'} \{r_{i'j} - \bar{r}_{i'}\}}{\sum_{i' \in \Omega_j} |w_{ii'}|}$$

Item Score vs. User Score

Item Score in Item-Item Collaborative Filtering:

$$s(i,j) = \bar{r}_j + \frac{\sum_{j' \in \Psi_i} w_{jj'} \{r_{ij'} - \bar{r}_{j'}\}}{\sum_{j' \in \Psi_i} |w_{jj'}|};$$

In which Ψ_i is the set of items that user i has rated and $w_{jj'}$ is called "Item Correlations" is as follows:

$$w_{jj'} = \frac{\sum_{i \in \Omega_{jj'}} (r_{ij} - \bar{r}_j)(r_{ij'} - \bar{r}_{j'})}{\sqrt{\sum_{i \in \Omega_{jj'}} (r_{ij} - \bar{r}_j)^2} \sqrt{\sum_{i \in \Omega_{jj'}} (r_{ij'} - \bar{r}_{j'})^2}}$$

Where Ω_j= users who rated item j.

$\Omega_{jj'}$= users who rated item j and item j'.

$\bar{r}_j$=average rating for item j.
spark.mllib supports model-based collaborative filtering, in which users and products are described by a small of latent factors that can be used to predict missing entries. Spark.mllib uses the ALS[1] algorithm to learn these latent factors.

2. **Matrix Factorization** says collecting all data is expensive. MF[2] allows you to find those features only by looking at the patterns between user's items and the ratings of those items

[1] Alternating Least Squares
[2] Matrix Factorization

(dimensionality reduction). This is like clustering concept. As with all unsupervised learning models, you can look at your data and observe what pattern you see.
So, Matrix Factorization is an unsupervised machine learning model.
Matrix Factorization Training:
To train the Matrix Factorization model, we assume that $\hat{R}$ is an approximation of R.

$$R \approx \hat{R} = WU^T$$

This is a form of linear regression.
First step of MF model: $\hat{R} = W^T U$

Simple example form of computing: let

$w_i = (1, 0.8, -1, 0.1, 1)$
$u_j = (1, 1.5, -1.3, 0, 1.2)$
$result = 1 * 1 + 0.8 * 1.5 + 1 * 1.3 + 0.1 * 0 + 1 * 1.2 = 4.7$

Matrix-Factorization-Training

How we train MF model? Since this is regression, we do that by sum of squared errors.
Squared error loss:

$$J = \sum_{i,j \in \Omega} \left(r_{ij} - \hat{r}_{ij}\right)^2 = \sum_{i,j \in \Omega} \left(r_{ij} - {w_i}^T u_j\right)^2$$

$\Omega = set\ of\ pairs\ (i, j)\ where\ user\ i\ rated\ product\ j.$
(We could also look at the mean squared error, but that just divides the dataset by the size, which is a constant and has no effect on the answer.)
The goal is to minimize the loss. We do that by finding the gradient, set to zero and solve it for the parameters w, u.
Solving for u:

$$\frac{\partial J}{\partial u_j} = 0 => u_j = \left(\sum_{i \in \Omega_j} w_i {w_i}^T\right)^{-1} \sum_{i \in \Omega_j} r_{ij} w_i$$

This seems to be like solving $u_j = A^{-1} b$.
Similarly we solve $J\ for\ w$. We set $\frac{\partial J}{\partial w_i} = 0$. So we get

$$\left(\sum_{j \in \Psi_i} u_j {u_j}^T\right) w_i = \sum_{j \in \Psi_i} r_{ij} u_j$$

This is like solving the equation $AX = b$. So $X = np.linalg.solve(A, b)$.

Expanding Matrix Factorization Model

Some of users are optimistic or pessimistic in rating. So there is a bias. Hence we write MF as a form of linear regression model regarding biases. In fact, we consider user's bias and product's bias and global average of ratings in the training set.

$$\widehat{r_{ij}} = w_i{}^T u_j + b_i + c_j + \mu$$

MF with Regularization.

3. Matrix Factorization vectorized

It's an alternative MF in numpy. It runs fastly, since it partially vectorized. In my machine, it takes about… per epoch….

4. Singular Value Decomposition (SVD) as an optional for semi vectorization MF:
   Those who are not comfortable with linear algebra can skip it, as it involves eigenvalue and eigenvector concepts.
   Notice SVD doesn't work when X, the decomposed matrix into U, S, and V^T, has missing values. In this case, we can not find the eigenvalues or eigenvectors of a matrix that has missing values.
   It seems better to refer the MF instead of SVD, as it is confusing.

There are also other better optionals to vectorize Matrix Factorization:

Deep Learning Methods:

i. Keras

ii. Deep Neural Network

5. Matrix Factorization in Keras
   We re-implement matrix factorization, but this time using the deep learning powerful library known as keras.
   - Keras is a Deep learning library with "automatic differentiation", which is especially helpful for working with large neural networks.
   - Algorithm for training neural networks is gradient descent.
     (Both gradient descent and alternating least square are valid training algorithms for matrix factorization.)

Note. Deriving gradient descent is easy, because we wouldn't have had to solve it for any of the parameters, but it is slow. Then reason we're not writing it in numpy is that it would be actually disadvantageous compared to keras. Keras take the advantage of GUI which leads to go very faster.

- One reason to get a better result with keras in comparison with before is that we split up data differently. This is by **embedding** the dataset.

Embedings Method

How do word embedding works and what does MF have to do with word embeddings?

Suppose that we have N words in our vocabulary and we correspond each word to a feature vector with dimension K. Then we'd have a N*K matrix. Whenever we want to get the vector for a particular word, the ith word for example, we just index that row in our word embedding matrix. It looks like the user matrix in matrix factorization.

Example. T-shirt → (0.5, -0.2, 1.8)

I	0.2	0.3	0.6
Like	0.8	0.7	0.9
This	0.1	0.6	0.6
T-shirt	0.5	-0.2	1.8

The idea behind word embeddings is that words are discrete objects and neural networks accept numbers (numerical feature vectors) as input. So, it doesn't make sense to pass a discrete object.

Word embeddings in keras

2 embedding layers:

```
emb_u= Embedding(N,k)

emb_m=Embedding(M,K)

u=Input()   #user

m=Input    #movie

eu=emb_u(u)      #return a k-size vector

em=emb_m(m)  #return a k-size vector

#dot them to get prediction:

r=dot(eu,em)
```

This is a model object. The next step is to choose an optimizer and loss function, which in our case is mean squared error.

Then we call just model.fit and this gives us back the loss per iteration which we can plot.

```
model=Model(inputs=[u,m], outputs=r)
```

model.compile(loss='mse',optimizer='sgd')

loss=model.fit(input_data, targets)

plot(loss)

the model has two inputs, but we have also bias.

Bias Terms

This isn't really straightforward. Keras is not a low level library like Theano and Tensorflow. So, you don't have access to objects like tensors, matrices, array, vector,…. Instead our level of abstraction is "layer". So we have embedding layer, dense layer, convolution layer, LSTM layer, etc. But the key to create bias term is just using embedding layer again.
Embedding(N,K) gives me a k-size vector for a user. Therefore, Embedding(N,1) gives me a scalar for a user. Exactly what the bias is!

```
#add user bias and movie bias:
bias_u=Embedding(N,1)
bias_m=Embedding(M,1)
#pass u and m through bias embeddings
bu=bias_u(u)
bm=bias_m(m)
r=add(r,bu,bm)    #r=dot(eu,em)
```

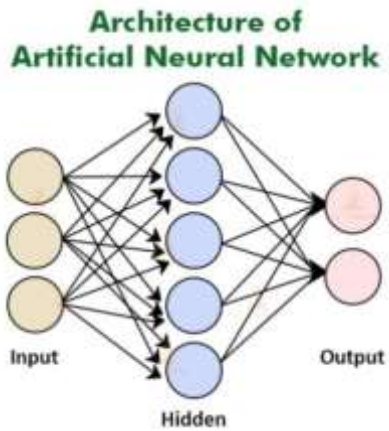
Global average:
We want the model:

$$r_{ij} = dot(w_i, u_j) + b_i + c_j + mu$$

Instead of having mu as part of the prediction, I subtract mu from the targets.
mu=target-actual rating

6. Deep Neural Network

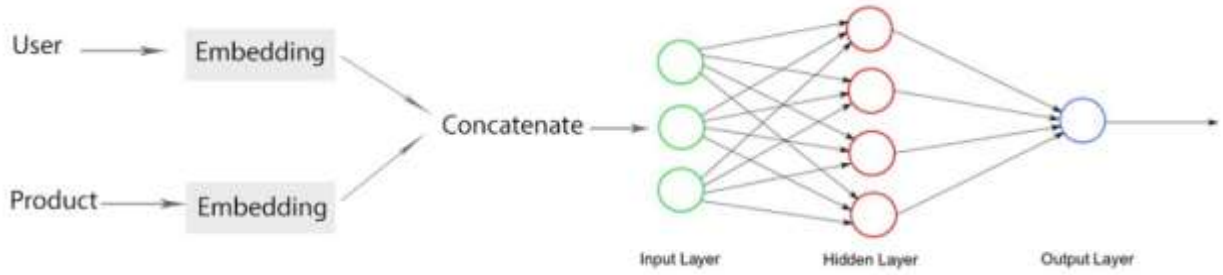
We can use deep learning instead of MF. Recall that the structure of a feedforward neural network.



We have some inputs about users and products which are not easy to gather this data. But Wait! We have told about embedding and feature vectors that present the user and feature vector that present the item.

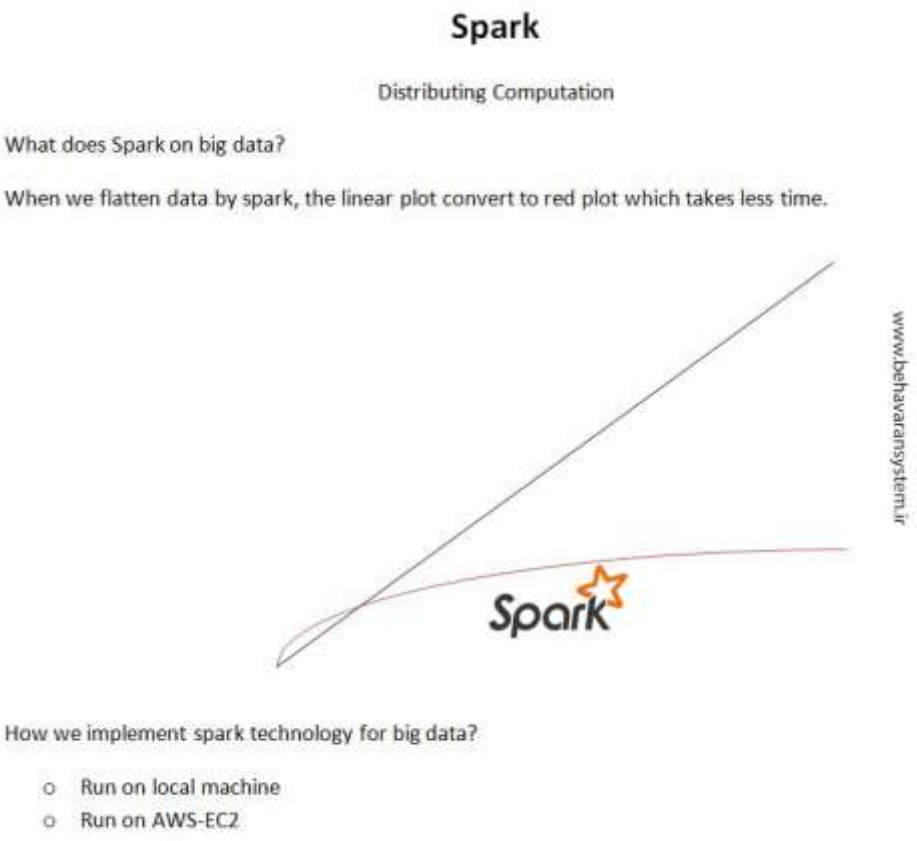
User Vector: W[i]

Product Vector: U[j]



We concatenate W[i] and U[j] together and assume it as input feature for neural network. Our model has just one hidden layer and since it is for regression, we have only one output.

7. Matrix Factorization on Spark



Explicit vs. implicit feedback

Although we see in many real-world use cases to only have access to implicit feedback (views, clicks, purchases, likes, shares etc.), the standard approach to matrix factorization-based collaborative filtering treats the entries in the user-item matrix as explicit. We use also explicit type. In fact, ALS.train() assumes training is explicit.