A Recommender System

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Obvious Applications

We are now advanced enough that we can aspire to a serious application. One of the most significant applications for some very large websites (Netflix, Amazon, etc.) are recommender systems.

“Customers who bought this product also bought these.”

“Here are some movies you might like…”

As well as many types of targeted advertising. However those of you with less commercial ambitions will find the core concepts here widely applicable to many types of data that require dimensionality reduction techniques.

Let’s go all Netflix

Netflix once (2009) had a $1,000,000 contest to with just this very problem[1]. We will start with a similar dataset. It looks like:

Movie Dataset (Movie ID, Title, Genre):
31,Dangerous Minds (1995),Drama
32,Twelve Monkeys (a.k.a. 12 Monkeys) (1995),Mystery|Sci-Fi|Thriller
34,Babe (1995),Children|Drama

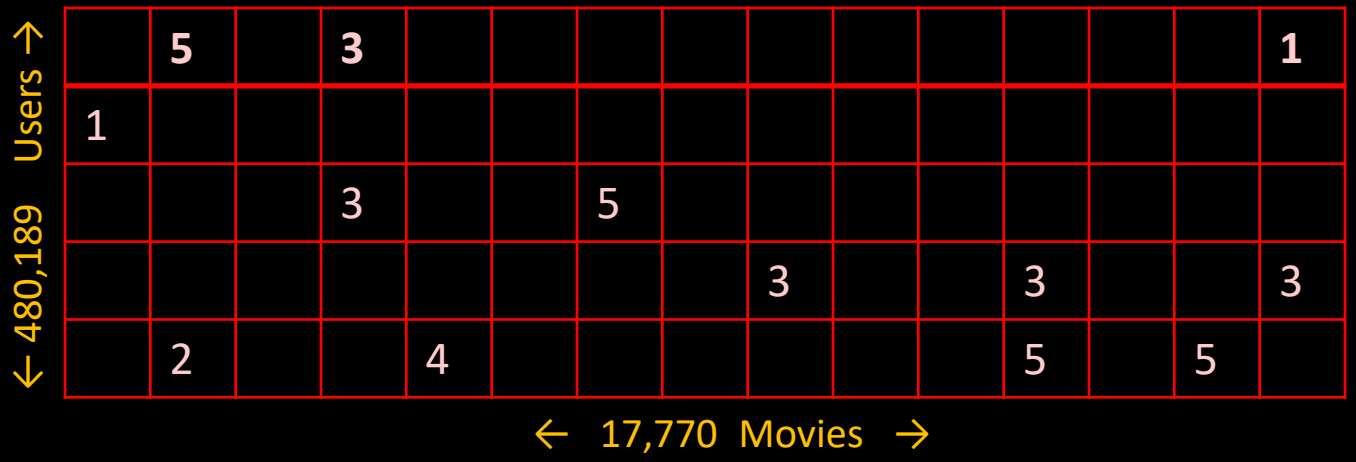
Ratings Dataset (User ID, Movie ID, Rating, Timestamp):
2,144,3.0,835356016
2,150,5.0,835355395
2,153,4.0,835355441
2,161,3.0,835355493

We won’t use the genres or timestamp fields for our analysis.

1) https://en.wikipedia.org/wiki/Netflix_Prize

Starting Point

What we are given is a large (100,480,507 ratings) and sparse (that is a little better than 1% of 8,532,958,530 matrix elements) list of ratings for users:



Objective

For any given user we would like to use their ratings, in combination with all the existing user ratings, to determine which movies they might prefer. For example, a user might really like *Annie Hall* and *The Purple Rose of Cairo* (both Woody Allen movies, although our database doesn't have that information). Can we infer from other users that they might like *Zelig*? That would be finding a latent variable. These might also include affinities for an actor, or director, or genre, etc. This type of algorithm is called *collaborative filtering*.

3	**5**	**2**	**3**	**5**	**2**	**4**	**3**	**1**	**4**	**3**	**1**	**3**	**2**	**1**
1	4	1	3	2	1	3	2	5	5	3	4	3	4	1
1	3	5	3	3	5	5	4	2	2	2	5	4	3	3
4	1	2	4	2	3	5	2	3	1	1	3	1	2	3
2	2	4	1	4	4	3	5	1	2	4	5	2	5	4

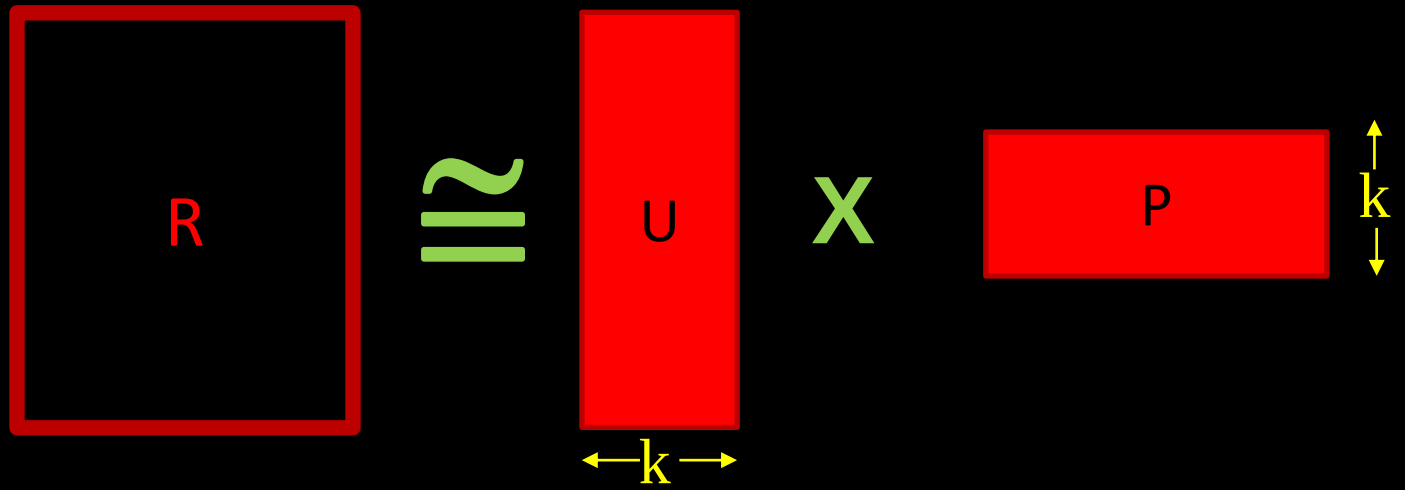
Users

Movies

Matrix Factorization

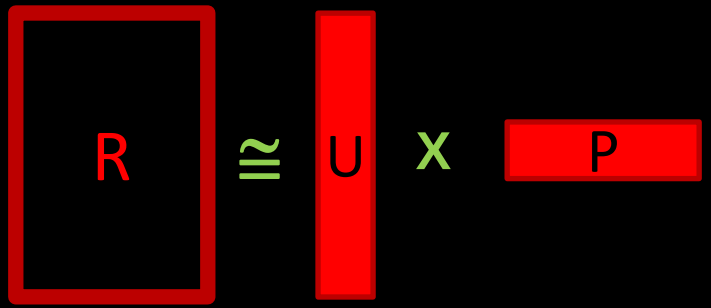
This resulting large, dense, matrix would be too big to actually keep around. We need to find a compressed representation where we can reproduce any given element we request. This will have to be lossy.

There are different ways to decompose a matrix. We will approximate our matrix as the product of two smaller matrices. The rank, k, of the new matrices will determine how accurate this approximation will be.



Lossy Compression Becomes Approximate Solution

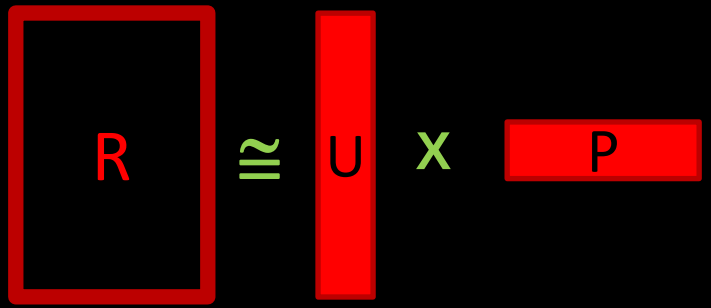
The process of lossy compressing the sparse R matrix is also going to provide us a means to construct its missing members (the dense matrix).



We will call our smaller matrices a *user feature matrix* and a *product feature matrix*. *This approximation is also going to smooth out the zeros and in the process give us our projected ratings.*

Why are we getting this two-for-one?

This provides an excellent introduction to a profound perspective on Machine Learning.



One way of thinking about learning is that we are compressing everything we know about the world into a smaller representation. Sometimes, but not usually, this can be seen explicitly, as here.

You can do this too.

Let's say you worked in a 1990's video store, but had never heard of Steven Spielberg. If you paid careful attention to the rental records you might notice that many people that rented *E.T.* also rented *Raiders of the Lost Ark* and *Jaws* and *Close Encounters* and *Jurassic Park*. So if a customer told you they really enjoyed an Indiana Jones movie, you might suggest they try *Jurassic Park*. All without knowing who the director was. You have inferred a hidden connection (latent effect).

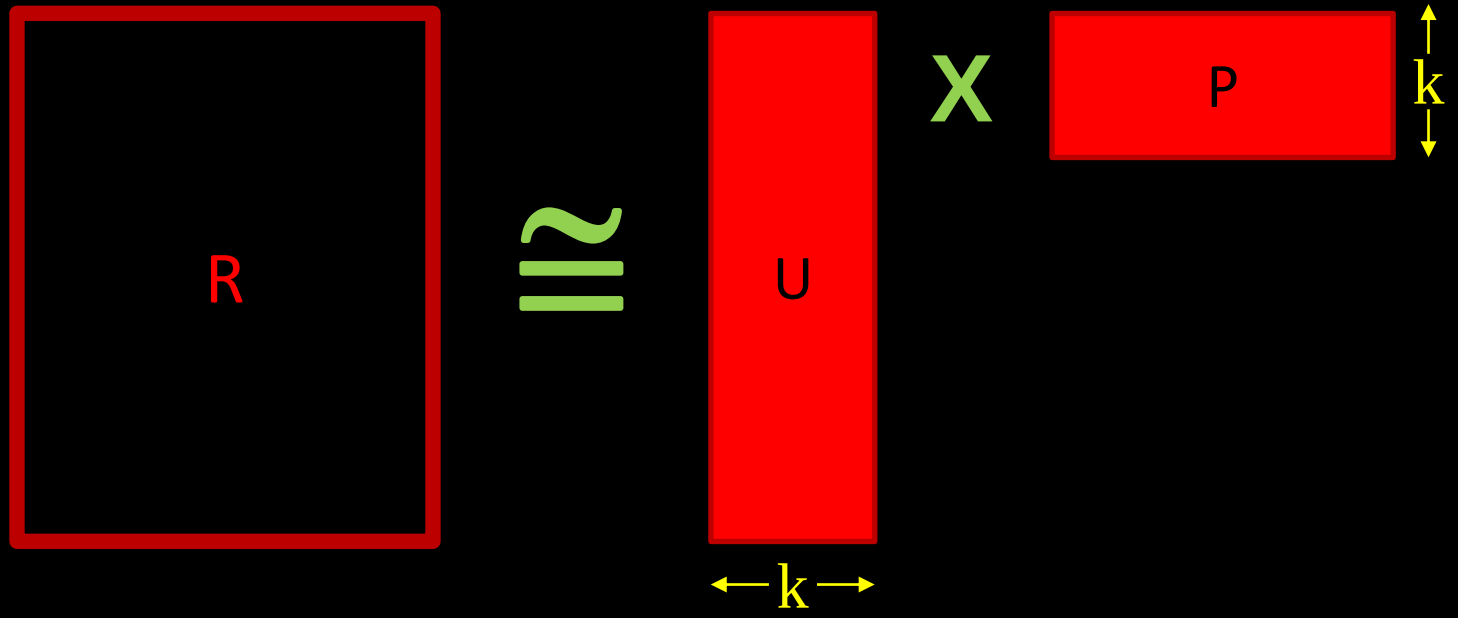
One can imaging many such hidden categories in our movie data: actors, genres, release dates, etc.

You can also imagine that the renters themselves possess these preferences hidden in their own data. Without it being explicitly noted, we might easily see that Mary likes documentaries and Joe loves movies with Cher.

We are thinking of reduced ways to represent these people ("likes documentaries") vs. the raw data!

Matrix Factorization

The rank k can now also be thought of as the number of *latent effects* we are incorporating. But it will not be as intuitively explicit as a simple category, and we will have to investigate an optimal size for this parameter.



Defining our error

In ML, defining the *error* (or *loss*, or *cost*) is often the core of defining the objective solution. Once we define the error, we can usually plug it into a canned solver which can minimize it. Defining the error can be obvious, or very subtle, or have multiple acceptable methods.

Clustering: For k-means we simply used the geometrical distance. It was actually the sum of the squared distances, but you get the idea.

Image Recognition: If our algorithm tags a picture of a cat as a dog, is that a larger error than if it tags it as a horse? Or a car? How would you quantify these?

Regression: Do you want to penalize a lot of medium errors more than an occasional large error?

Recommender: We will take the Mean Square Error distance between our given matrix and our approximation as a starting point.

Mean Square Error plus Regularization

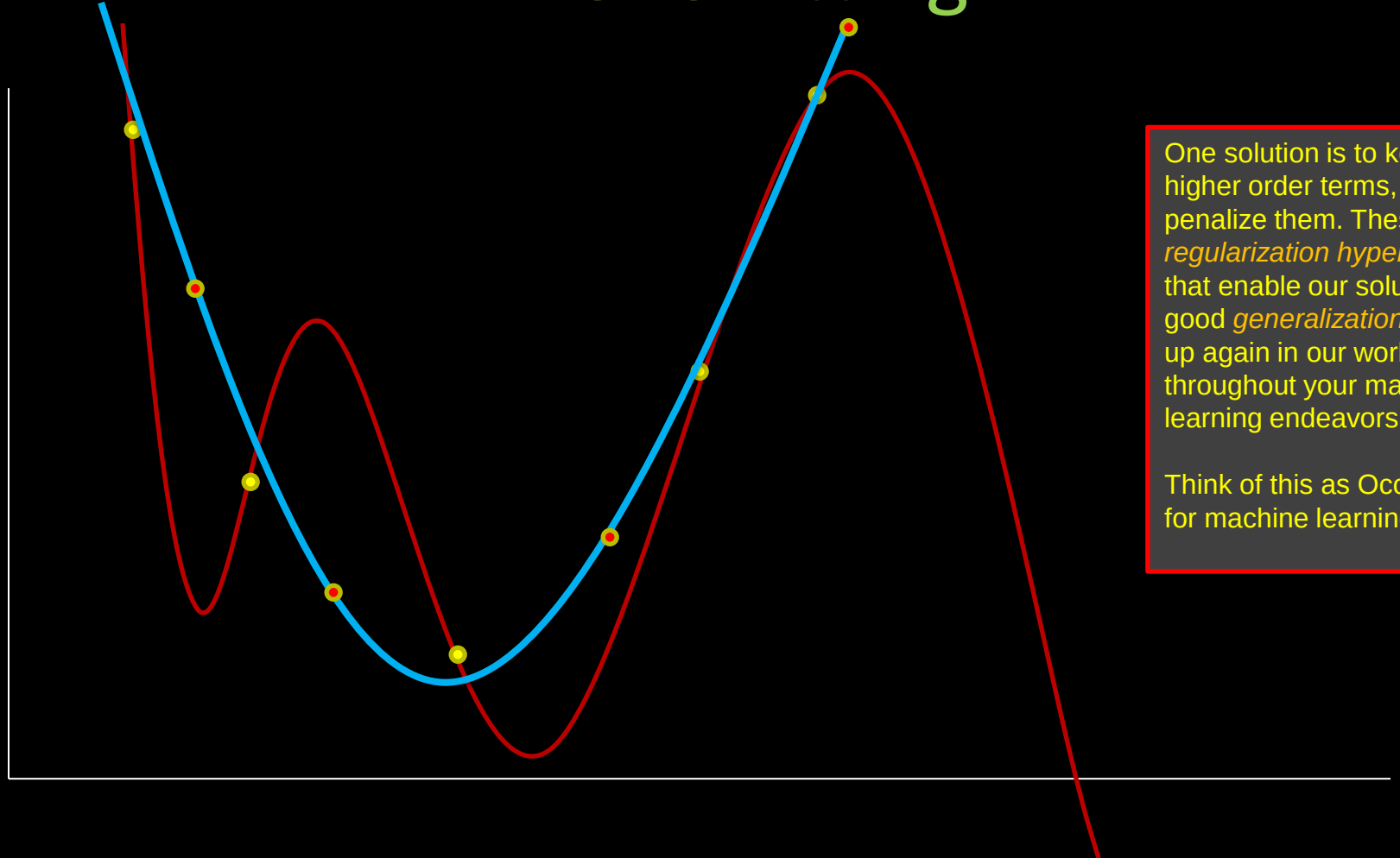
We will also add a term to discourage *overfitting* by damping large elements of U or P. This is called *regularization* and versions appear frequently in error functions.

$$\text{Error} = | R - UxP |^2 + \lambda(\text{Penalty for large elements})$$

The $||$ notation means "sum the squares of all the elements and then take the square root".

You may wonder how we can have "too little" error – the pursuit of which leads to overfitting. Think back to our clustering problem. We could drive the error as low as we wanted by adding more clusters (up to 5000!). But we weren't really finding new clusters. Variations of this phenomena occur throughout machine learning.

Overfitting



One solution is to keep using higher order terms, but to penalize them. These *regularization hyperparameters* that enable our solution to have good *generalization* will show up again in our workshop, and throughout your machine learning endeavors.

Think of this as Occam's Razor for machine learning.

Mean Square Error plus Regularization

Here is exactly our error term with regularization. MLLIB scales this factor for us based on the number of ratings (this tweak is called ALS-WR).

$$\text{Error} = | R - UxP |^2 + \lambda(|U|^2 + |P|^2)$$

The $||$ notation means "sum the squares of all the elements and then take the square root".

Additionally, we need to account for our missing (unrated) values. We just zero out those terms. Here it is term-by-term:

$$\text{Error} = \Sigma_{I,j}\, w_{I,j}\, (R_{I,j} - (UxP)_{ij})^2 + \lambda(|U|^2 + |P|^2) \qquad w_{I,j} = 0 \text{ if } R_{I,j} \text{ is unknown}$$

Note that we now have two hyperparameters, k and λ, that we need to select intelligently.

Alternating Least Squares

To actually find the U and P that minimize this error we need a solving algorithm.

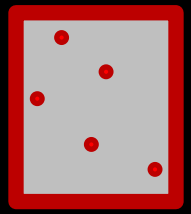
SGD, a go-to for many ML problems and one we will use later, is not quite as efficient for billions of sparse parameters, which we can easily reach with these types of problems. We are dealing with *Users X Items* elements here.

Instead we use Alternating Least Squares (ALS), also built into MLLIB.

- Alternating least squares cheats by holding one of the arrays constant and then doing a classic least squares fit on the other array parameters. Then it does this for the other array.
- This is easily parallelized.
- It works well with sparse inputs. The algorithm scales linearly with observed entries.

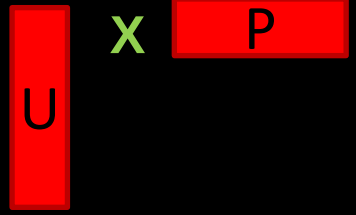
Here Is Our Plan

Given Ratings



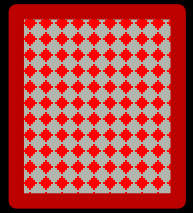
≅

Use ALS

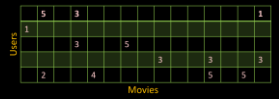


=

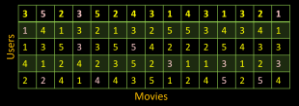
Predicted Ratings



Sparse



Dense



Training, Validation and Test Data

We use the training data to create our solution, the UxP matrix here.

The validation data is used to verify we are not overfitting: to stop training after enough iterations, to adjust λ or k here, or to optimize the many other *hyperparameters* you will encounter in ML.

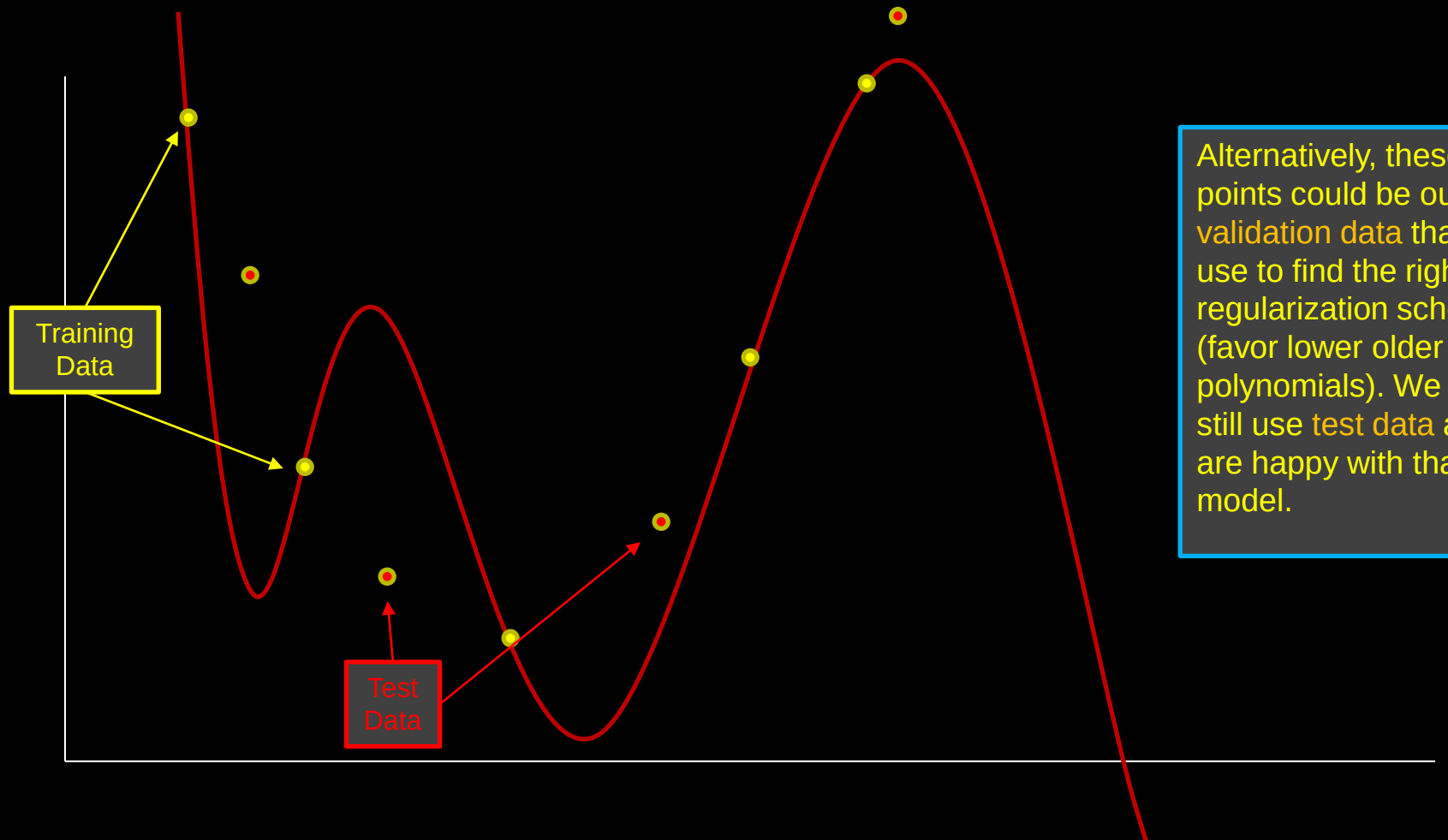
The test data must be saved to judge our final solution.

Reusing, or subtly mixing, the training, validation and t
confusion.

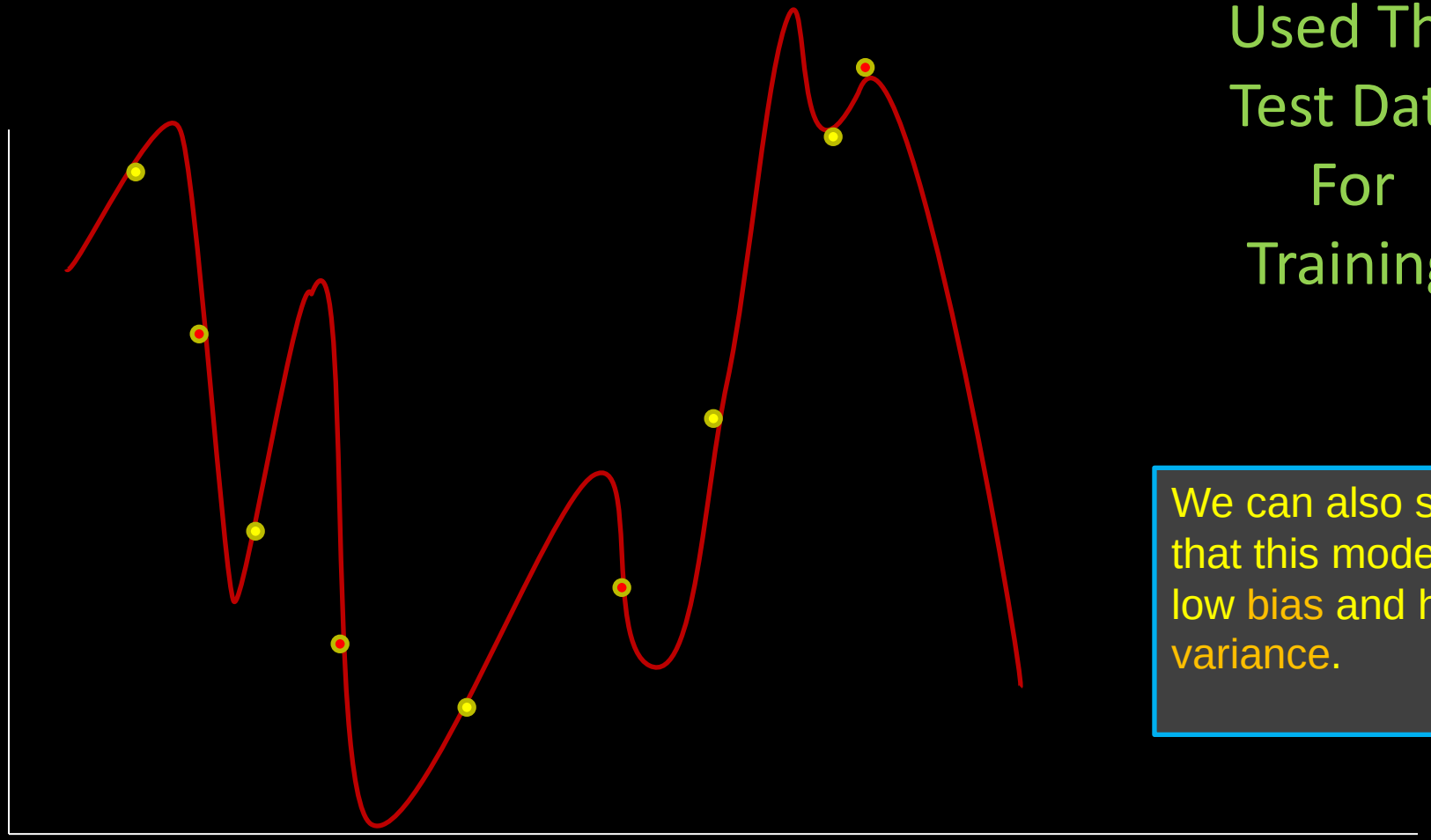
What proportions of your data to use for each of these
might want to start by copying from similar work or exa

There are techniques to slice-and-cycle share the training and validation data, called *cross-validation*. Don't try this with the test data!

Reality Check By Test Data



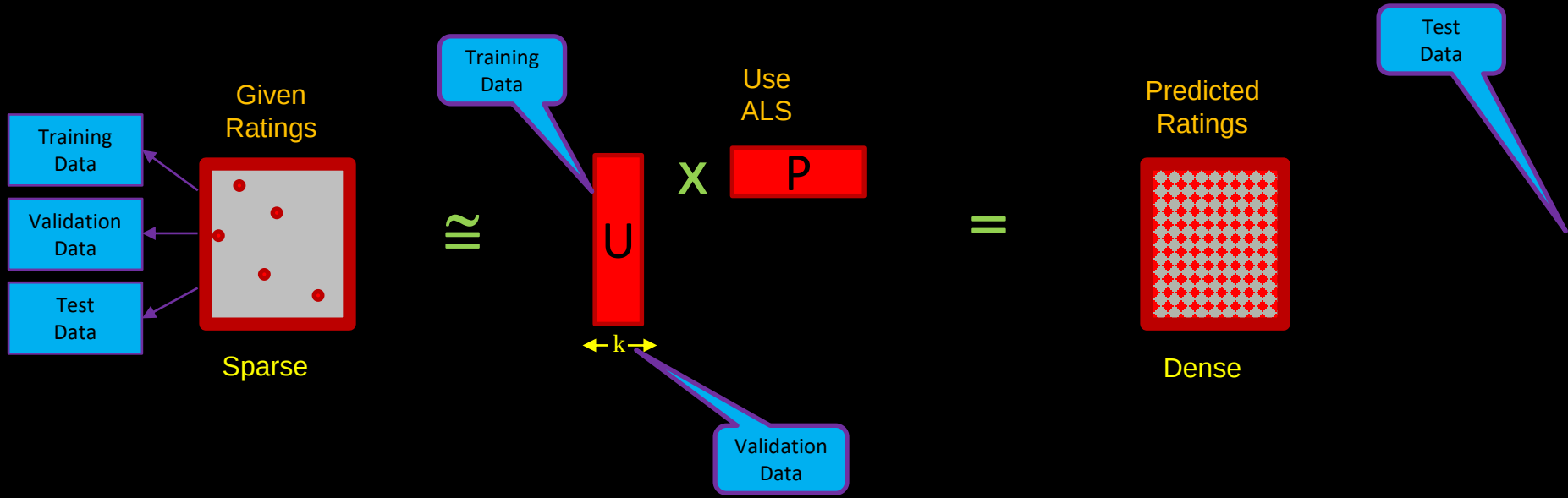
Alternatively, these new points could be our validation data that we use to find the right regularization scheme (favor lower older polynomials). We would still use test data after we are happy with that model.



Used The Test Data For Training

We can also say that this model has low bias and high variance.

Where does our data come into play?



Let’s Build A Recommender

We have all the tools we need, so let’s fire up PySpark and create a scalable recommender. Our plan is:

1. Load and parse data files
2. Create ALS model
3. Train it with varying ranks (k) to find reasonable hyperparameters

4. Add a new user
5. Get top recommendations for new user

Building a Recommender

```
login06% interact
...
r288%
r288% module load spark
r288% pyspark
```

```
Using Python version 3.7.4 (de
SparkSession available as 'spa
>>>
>>> ratings_raw_RDD = sc.textF
>>> ratings_RDD = ratings_raw_
>>>
>>> training_RDD, validation_R
>>>
>>> predict_validation_RDD = v
>>> predict_test_RDD = test_RD
```

```
),float(tokens[2])))
```

```
>>> training_RDD.take(4)
[(1, 1029, 3.0), (1, 1061, 3.0), (1, 1263, 2.0), (1, 1371, 2.5)]
>>> predict_validation_RDD.take(4)
[(1, 1129), (1, 1172), (1, 1405), (1, 2105)]
>>>
```

ovie,rating) data.

Ds.

data for our prediction RDDs.

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
>>> ratings_raw_RDD = sc.textFile('ratings.csv')
>>> ratings_RDD = ratings_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),int(tokens[1]),float(tokens[2])))
>>>
>>> training_RDD, validation_RDD, test_RDD = ratings_RDD.randomSplit([3, 1, 1], 0)
>>>
>>> predict_validation_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
>>> predict_test_RDD = test_RDD.map(lambda x: (x[0], x[1]))
>>>
>>>
>>> from pyspark.mllib.recommendation import ALS
>>> import math
>>>
>>> seed = 5
>>> iterations = 10
>>> regularization = 0.1
>>> trial_ranks = [4, 8, 12]
>>> lowest_error = float('inf')
```

Import mllib and set some variables we are about to use.

```
>>> ratings_raw_RDD = sc.textFile('ratings.csv')
>>> ratings_RDD = ratings_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),int(tokens[1]),float(tokens[2])))
>>>
>>> training_RDD, validation_RDD, test_RDD = ratings_RDD.randomSplit([3, 1, 1], 0)
>>>
>>> predict_validation_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
>>> predict_test_RDD = test_RDD.map(lambda x: (x[0], x[1]))
>>>
>>>
>>> from pyspark.mllib.recommendation import ALS
>>> import math
>>>
>>> seed = 5
>>> iterations = 10
>>> regularization = 0.1
>>> trial_ranks = [4, 8, 12]
>>> lowest_error = float('inf')
>>>
>>> for k in trial_ranks:
...     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
...     #Coercing ((u,p),r) tuple format to accomodate join
...     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
...     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
...     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
...     print ('For k=',k,'the RMSE is', error)
...     if error < lowest_error:
...         best_k = k
...         lowest_error = error
...
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
>>>
>>> print('The best rank is size', best_k)
The best rank is size 4
```

Run our ALS model on various ranks to see which is best.

```
>>> for k in trial_ranks:
>>>    model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>    #Coercing ((u,p),r) tuple format to accomodate join
>>>    predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>    ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>    error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>    print ('For k=',k,'the RMSE is', error)
>>>    if error < lowest_error:
>>>        best_k = k
>>>        lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
```

The ALS.train() routines gives us:

```
>>> model.predictAll(predict_validation_RDD).take(2)
[Rating(user=463, product=4844, rating=2.7640960482284322), Rating(user=380, product=4844, rating=2.399938320644199)]
```

To do the "RMS error" math, we want elements with a *(Given,Predicted)* value for each *(User,Movie)* key:

```
>>> ratings_and_preds_RDD.take(2)
[((119, 145), (4.0, 2.903215714486778)), ((407, 5995), (4.5, 4.604779028840272))]
```

So the next two lines get us from *here to there*.

```
>>> for k in trial_ranks:
>>>     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>     #Coercing ((u,p),r) tuple format to accomodate join
>>>     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>     print ('For k=',k,'the RMSE is', error)
>>>     if error < lowest_error:
>>>         best_k = k
>>>         lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
```

```
>>> model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((463, 4844), 2.7640960482284322), ((380, 4844), 2.399938320644199)]
```

That map gets us to a pair RDD with *[(User,Movie), rating]* format.

Now do this with the validation RDD:

```
>>> validation_RDD.take(2)
[(1, 1129, 2.0), (1, 1172, 4.0)]
>>>
>>> validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((1, 1129), 2.0), ((1, 1172), 4.0)]
```

```
>>> for k in trial_ranks:
>>>    model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>    #Coercing ((u,p),r) tuple format to accomodate join
>>>    predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>    ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>    error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>    print ('For k=',k,'the RMSE is', error)
>>>    if error < lowest_error:
>>>        best_k = k
>>>        lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
```

To collect rating values for common (User,Movie) keys calls for a join()

Data before join:

```
>>> predictions_RDD.take(2)
[((463, 4844), 2.7640960482284322), ((380, 4844), 2.399938320644199)]
>>>
>>> validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((1, 1129), 2.0), ((1, 1172), 4.0)]
```

Results of join:

```
>>> ratings_and_preds_RDD.take(2)
[((119, 145), (4.0, 2.903215714486778)), ((407, 5995), (4.5, 4.604779028840272))]
```

```
>>> for k in trial_ranks:
>>>    model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>    #Coercing ((u,p),r) tuple format to accomodate join
>>>    predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>    ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>    error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>    print ('For k=',k,'the RMSE is', error)
>>>    if error < lowest_error:
>>>        best_k = k
>>>        lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
>>>
>>> print('The best rank is size', best_k)
The best rank is size 4
>>>
>>> model = ALS.train(training_RDD, best_k, seed=seed, iterations=iterations, lambda_=regularization)
>>> predictions_RDD = model.predictAll(predict_test_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>> ratings_and_preds_RDD = test_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>> error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>> print ('For testing data the RMSE is %s' % (error))
For testing data the RMSE is 0.9406803213698973
```

This is our fully tested model (smallest dataset).
These results were reported against the test_RDD.

Adding a User

```
.
.
.
>>>
>>> new_user_ID = 0
>>> new_user = [
        (0,100,4), # City Hall (1996)
        (0,237,1), # Forget Paris (1995)
        (0,44,4),  # Mortal Kombat (1995)
        (0,25,5),  # etc....
        (0,456,3),
        (0,849,3),
        (0,778,2),
        (0,909,3),
        (0,478,5),
        (0,248,4)
       ]
>>>
>>> new_user_RDD = sc.parallelize(new_user)
>>>
>>> updated_ratings_RDD = ratings_RDD.union(new_user_RDD)
>>>
>>> updated_model = ALS.train(updated_ratings_RDD, best_rank, seed=seed, iterations=iterations,
lambda_=regularization)
>>>
```

I checked that ID 0 is unused with a quick
`ratings_RDD.filter(lambda x: x[0]=='0').count()"`

Note that we are joining, and then training, with ALL data now - the ratings RDD. We are confident we know what we are doing and are done testing.

This need to retrain on all the data even though we just added a single user is call a *cold-start* problem. We might patch over that with a different technique like a Deep Learning *content based recommender* system. Most serious recommenders use an *ensemble* of algorithms.

Let’s get some predictions…

```
.
.
.
>>>
>>> movies_raw_RDD = sc.textFile('movies.csv')
>>> movies_RDD = movies_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),tokens[1]))
>>>
>>> new_user_rated_movie_ids = map(lambda x: x[1], new_user)
>>> new_user_unrated_movies_RDD = movies_RDD.filter(lambda x: x[0] not in new_user_rated_movie_ids).map(lambda x: (new_user_ID, x[0]))
>>> new_user_recommendations_RDD = updated_model.predictAll(new_user_unrated_movies_RDD)
```

```
>>> new_user_unrated_movies_RDD.take(3)
[(0, 1), (0, 2), (0, 3)]
>>> new_user_recommendations_RDD.take(2)
[Rating(user=0, product=4704, rating=3.606560950463134), Rating(user=0, product=4844, rating=2.1368358868224036)]
```

Let see some titles

```
.
.
.
>>>
>>> product_rating_RDD = new_user_recommendations_RDD.map(lambda x: (x.product, x.rating))
>>> new_user_recommendations_titled_RDD = product_rating_RDD.join(movies_RDD)
>>> new_user_recommendations_formatted_RDD = new_user_recommendations_titled_RDD.map(lambda x: (x[1][1],x[1][0]))
>>>
>>> top_recomends = new_user_recommendations_formatted_RDD.takeOrdered(10, key=lambda x: -x[1])
>>> for line in top_recomends: print (line)
...
('Maelstr\xf6m (2000)', 6.2119957527973355)
('King Is Alive', 6.2119957527973355)
('Innocence (2000)', 6.2119957527973355)
('Dangerous Beauty (1998)', 6.189751978239315)
('Bad and the Beautiful', 6.005879185976944)
('Taste of Cherry (Ta'm e guilass) (1997)', 5.960748198887891)
('The Lair of the White Worm (1988)', 5.958594728894122)
('Mifune's Last Song (Mifunes sidste sang) (1999)', 5.934820295566816)
('Business of Strangers', 5.899232655788708)
>>>
>>> one_movie_RDD = sc.parallelize([(0, 800)]) # Lone Star (1996)
>>> rating_RDD = updated_model.predictAll(one_movie_RDD)
>>> rating_RDD.take(1)
[Rating(user=0, product=800, rating=4.100848893773136)]
```

Looks like we can sort by value after all!

Behind the scenes takeOrdered() just does the key/value swap and SortByKey that we previously did ourselves.

```
>>> new_user_recommendations_titled_RDD.take(2)
[(111360, (1.0666741148393921, 'Lucy (2014)')), (49530, (1.8020006042285814, 'Blood Diamond (2006)'))]
>>> new_user_recommendations_formatted_RDD.take(2)
[('Lucy (2014)', 1.0666741148393921), ('Blood Diamond (2006)', 1.8020006042285814)]
```

Exercises

1) We noticed that out top ranked movies have ratings higher than 5. This makes perfect sense as there is no ceiling implied in our algorithm and one can imagine that certain combinations of factors would combine to create “better than anything you’ve seen yet” ratings.

Maybe you have a friend that really likes Anime. Many of her ratings for Anime are 5. And she really likes Scarlett Johansson and gives her movies lots of 5s. Wouldn’t it be fair to consider her rating for *Ghost in the Shell* to be a 7/5?

Nevertheless, we may have to constrain our ratings to a 1-5 range. Can you normalize the output from our recommender such that our new users only sees ratings in that range?

2) We haven’t really investigated our convergence rate. We specify 10 iterations, but is that reasonable? Graph your error against iterations and see if that is a good number.

3) I mentioned that our larger dataset does benefit from a rank of 12 instead of 4 (as one might expect). The larger datasets (ratings-large.csv and movies-large.csv) are available to you in ~training/LargeMovies. Prove that the error is less with a larger rank. How does this dataset benefit from more iterations? Is it more effective to spend the computation cycles on more iterations or larger ranks?

4) We could have used the very similar *pyspark.ml.recommendation* API, which uses dataframes. It requires a little more type checking, so we used the classic RDD API *pyspark.mllib.recommendation* instead - for conciseness. Try porting this example to that API. Is this a better way to work?