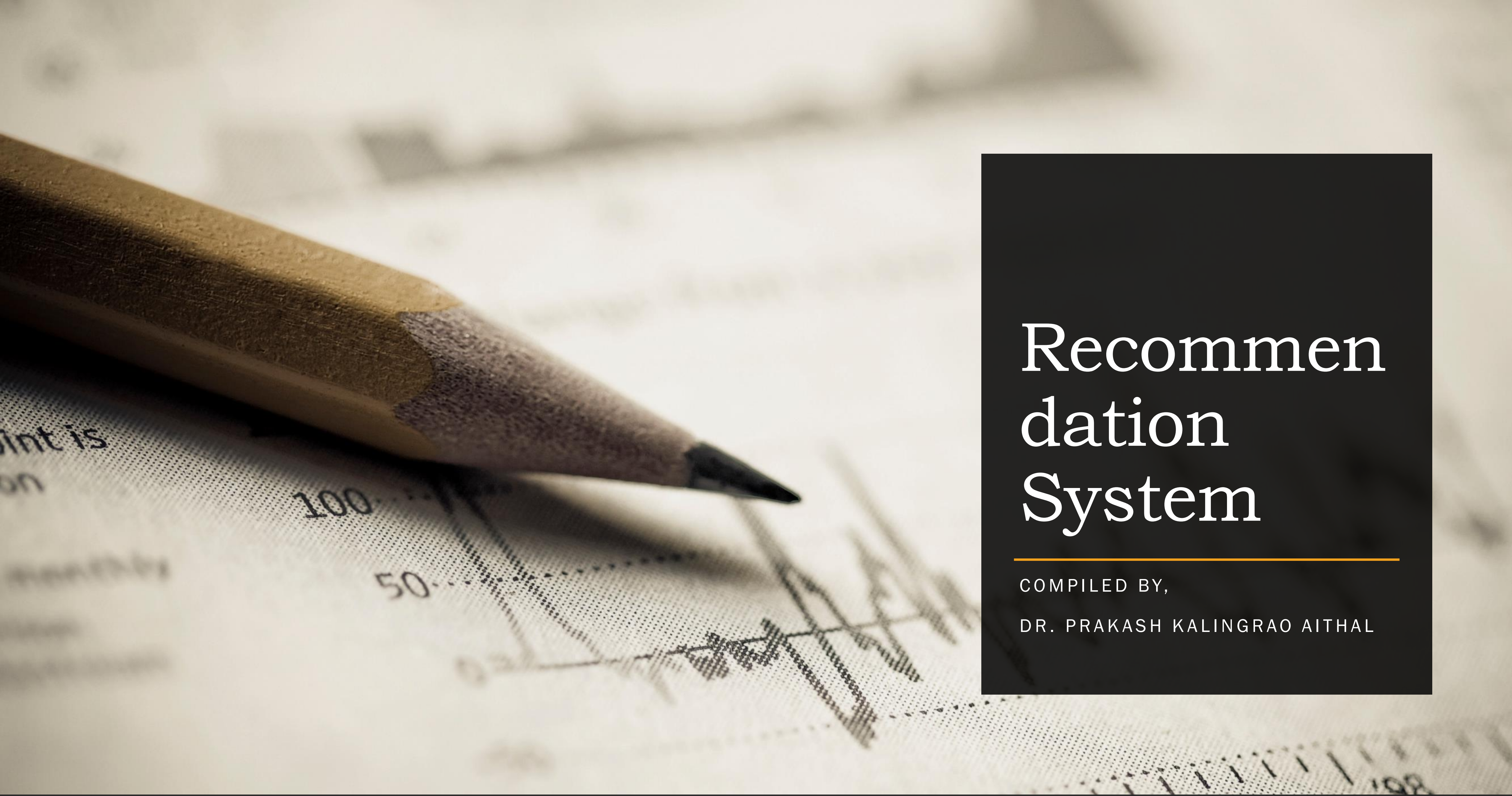


Recommendation System

COMPILED BY,

DR. PRAKASH KALINGRAO AITHAL

Requirement of Recommendation System

Q: List the consideration while making a Recommendation System

We need to choose a recommender algorithm that is suitable for our data. Here are our considerations:

Implicit feedback

The data is comprised entirely of interactions between users and artists' songs.It contains no information about the users or about the artists other than their names. We need an algorithm that learns without access to user or artist attributes. These are typically called collaborative filtering algorithms. For example, deciding that two users might share similar tastes because they are the same age *is not* an example of collaborative filtering. Deciding that two users might both like the same song because they play many other songs that are the same *is* an example.

Requirement of Recommendation System

Sparsity

Our dataset looks large because it contains tens of millions of play counts. But in a different sense, it is small and skimpy, because it is sparse. On average, each user has played songs from about 171 artists—out of 1.6 million. Some users have listened to only one artist. We need an algorithm that can provide decent recommendations to even these users. After all, every single listener must have started with just one play at some point!

Requirement of Recommendation System

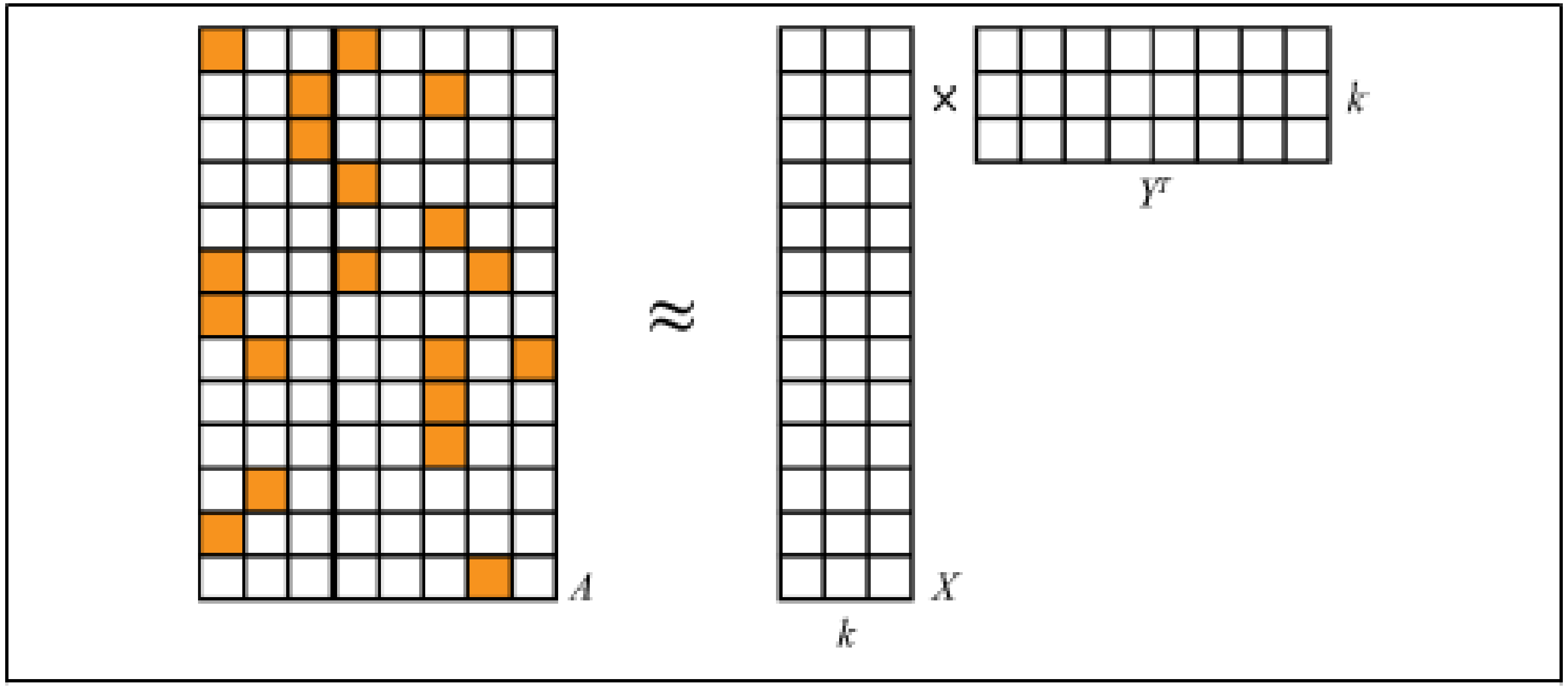
Scalability and real-time predictions

Finally, we need an algorithm that scales, both in its ability to build large models and to create recommendations quickly. Recommendations are typically required in near real time—within a second, not tomorrow.

Matrix Factorization

Mathematically, these algorithms treat the user and product data as if it were a large matrix *A*, where the entry at row *i* and column *j* exists if user *i* has played artist *j*. *A* is sparse: most entries of *A* are 0, because only a few of all possible user-artist combinations actually appear in the data. They factor *A* as the matrix product of two smaller matrices, *X* and *Y*. They are very skinny—both have many rows because *A* has many rows and columns, but both have just a few columns (*k*). The *k* columns correspond to the latent factors that are being used to explain the interaction data.

Matrix Factorization



what are matrix completion algos?

Matrix Factorization

These algorithms are sometimes called matrix completion algorithms, because the original matrix *A* may be quite sparse, but the product *XYT* is dense. Very few, if any, entries are 0, and therefore the model is only an approximation of *A*. It is a model in the sense that it produces ("completes") a value for even the many entries that are missing (that is, 0) in the original *A*.

Matrix Factorization

This is a case where, happily, the linear algebra maps directly and elegantly to intuition. These two matrices contain a row for each user and each artist. The rows have few values—*k*. Each value corresponds to a latent feature in the model. So the rows express how much users and artists associate with these *k* latent features, which might correspond to tastes or genres. And it is simply the product of a user-feature and feature-artist matrix that yields a complete estimation of the entire, dense user-artist interaction matrix. This product might be thought of as mapping items to their attributes and then weighting those by user attributes.

Matrix Factorization

The bad news is that $A = XYT$ generally has no exact solution at all, because X and Y aren't large enough (technically speaking, too low rank) to perfectly represent A. This is actually a good thing. A is just a tiny sample of all interactions that *could* happen. In a way, we believe A is a terribly spotty and therefore hard-to-explain view of a simpler underlying reality that is well explained by just some small number of factors, k, of them. Think of a jigsaw puzzle depicting a cat. The final puzzle is simple to describe: a cat. When you're holding just a few pieces, however, the picture you see is quite difficult to describe.

Matrix Factorization

XYT should still be as close to A as possible. After all, it’s all we’ve got to go on. It will not and should not reproduce it exactly. The bad news again is that this can’t be solved directly for both the best X and best Y at the same time. The good news is that it’s trivial to solve for the best X if Y is known, and vice versa. But neither is known beforehand!

Alternating Least Squares Algorithm

With ALS, we will treat our input data as a large, sparse matrix *A*, and find out *X* and *Y*, as discussed in previous section. At the start, *Y* isn't known, but it can be initialized to a matrix full of randomly chosen row vectors. Then simple linear algebra gives the best solution for *X*, given *A* and *Y*. In fact, it's trivial to compute each row *i* of *X* separately as a function of *Y* and of one row of *A*. Because it can be done separately, it can be done in parallel, and that is an excellent property for large-scale computation:

$$A_i Y \left(Y^T Y \right)^{-1} = X_i$$

Alternating Least Squares Algorithm

Equality can’t be achieved exactly, so in fact the goal is to minimize $|AiY(YTY)-1 - Xi|$, or the sum of squared differences between the two matrices’ entries. This is where the “least squares” in the name comes from. In practice, this is never solved by actually computing inverses, but faster and more directly via methods like the QR decomposition. This equation simply elaborates on the theory of how the row vector is computed. The same thing can be done to compute each *Yj* from *X*. And again, to compute *X* from *Y*, and so on. This is where the “alternating” part comes from. There’s just one small problem: *Y* was made up—and random! *X* was computed optimally, yes, but gives a bogus solution for *Y*. Fortunately, if this process is repeated, *X* and *Y* do eventually converge to decent solutions.

Alternating Least Squares Algorithm

When used to factor a matrix representing implicit data, there is a little more complexity to the ALS factorization. It is not factoring the input matrix *A* directly, but a matrix *P* of 0s and 1s, containing 1 where *A* contains a positive value and 0 elsewhere. The values in *A* are incorporated later as weights. This detail is beyond the scope of this book but is not necessary to understand how to use the algorithm.

Finally, the ALS algorithm can take advantage of the sparsity of the input data as well. This, and its reliance on simple, optimized linear algebra and its data-parallel nature, make it very fast at large scale.

Recommendation System

Data Preparation:

- Load your data into a Spark DataFrame. This could be user-item interaction data, such as ratings or purchases, or any other relevant data for your recommendation task.
- Preprocess your data as needed. This might include handling missing values, converting categorical variables to numerical format, or other data transformations.

Recommendation System

Model Training:

Choose a recommendation algorithm. PySpark provides several built-in algorithms for collaborative filtering, such as Alternating Least Squares (ALS) and Singular Value Decomposition (SVD).

Train your model using the fit() method of the chosen algorithm. This involves specifying the user and item columns, and optionally a rating column if you're using explicit feedback.

Recommendation System

Model Evaluation:

- Evaluate your model using appropriate metrics such as RMSE (Root Mean Squared Error) for explicit feedback or MAP (Mean Average Precision) for implicit feedback.
- Tune your model hyperparameters as needed to improve performance.

Recommendation System

Model Deployment:

Once you're satisfied with your model's performance, you can deploy it to make recommendations in real-time or in batch mode.

For real-time recommendations, you can use the transform() method of your trained model to generate recommendations for a specific user or item.

For batch recommendations, you can use the recommendForAllUsers() or recommendForAllItems() methods to generate recommendations for all users or items.

Recommendation System

Monitoring and Maintenance:

- Monitor your recommendation system's performance over time and make adjustments as needed.
- Re-train your model periodically with new data to keep it up-to-date.

Matrix Factorization

Matrix factorization is a mathematical technique used in various fields, including machine learning and data analysis, to decompose a matrix into two or more matrices that represent the original matrix in a more compact form. This technique is particularly popular in recommendation systems, where it is used to model user-item interactions.

In the context of recommendation systems, matrix factorization is often used to represent the user-item interaction matrix as the product of two lower-dimensional matrices: a user matrix and an item matrix. Each row of the user matrix represents a user, and each column of the item matrix represents an item. The elements of these matrices represent the latent factors (or features) associated with users and items.

Matrix Factorization

The goal of matrix factorization in recommendation systems is to learn these latent factors in such a way that the product of the user and item matrices closely approximates the original user-item interaction matrix. This allows the recommendation system to predict the likelihood of a user interacting with an item that they have not yet interacted with.

Advantages of Matrix Factorization

Matrix factorization has several advantages in recommendation systems, including:

1. **Scalability**: Matrix factorization can handle large, sparse matrices efficiently, making it suitable for recommendation systems with a large number of users and items.
2. **Flexibility**: Matrix factorization can capture complex relationships between users and items by learning latent factors that represent user preferences and item attributes.
3. **Cold Start Handling**: Matrix factorization can handle cold start problems, where new users or items with little to no interaction data can still be recommended based on their latent factors.
4. **Personalization**: Matrix factorization allows for personalized recommendations by learning user-specific latent factors.

ALS Algorithm

```
als = ALS(
    userCol="user_id",
    itemCol="product_id",
    ratingCol="score",
)
```

acronym:
rank max reg al

ALS Algorithm Hyperparameters

setRank(10)

The number of latent factors in the model, or equivalently, the number of columns *k* in the user-feature and product-feature matrices. In nontrivial cases, this is also their rank.

setMaxIter(5)

The number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.

setRegParam(0.01)

A standard overfitting parameter, also usually called *lambda*. Higher values resist overfitting, but values that are too high hurt the factorization's accuracy.

setAlpha(1.0)

Controls the relative weight of observed versus unobserved user-product interactions in the factorization.

ParamGridBuilder

The ParamGridBuilder class in PySpark's ml.tuning module is used to construct a grid of hyperparameters for tuning a machine learning model. This grid is then used in conjunction with a CrossValidator or TrainValidationSplit to search for the best combination of hyperparameters based on some evaluation metric.

The ParamGridBuilder is used to construct a grid of hyperparameters for tuning a machine learning model. Each hyperparameter is specified as a key-value pair, where the key is the name of the hyperparameter and the value is a list of possible values to search over. The build() method is then used to construct the grid of hyperparameters.

ParamGridBuilder

```
from pyspark.ml.tuning import CrossValidator,  ParamGridBuilder
parameter_grid = (
    ParamGridBuilder()
    .addGrid(als.rank, [1, 5])
    .addGrid(als.maxIter, [20])
    .addGrid(als.regParam, [0.05])
    .addGrid(als.alpha,[1])
    .build()
)
```

References

Sandya Ryza, Uri Laserson, Sean Owen and Josh Wills, Advanced Analytics with Spark (2e), O’Reilly Media Inc, 2017