Big Data Analytics

Chapter 3: Recommender Systems via Matrix Factorization

Following: [3] "**Advanced Analytics with Spark**", Chapter 3

Prof. Dr. Martin Theobald

Faculty of Science, Technology & Communication

University of Luxembourg

What is a Recommender System?

Recommender systems are at the intersection of data mining and predictive analysis, in which the goal is to predict a rating that a user would give to a (previously unrated) item.

- **Collaborative filtering** aims to find recommendations either based on the users' past behavior, or on the similarity of the user to other users who have previously rated items.
- **Content-based filtering** aims to find recommendations based on the similarity of an item to other items that the same user has rated.
- These two aspects are often combined…

Most "famous" example of a recommender system: http://www.netflixprize.com, which awarded 1 Mio USD to the winning team in 2009.

- The most accurate algorithm in 2007 used an ensemble method of 107 different algorithmic approaches, which were blended into a single prediction.
- The Netflix data set consisted of 100 Mio movie ratings of the following structure:

```
<user_id, movie_id, date_of_rating, rating>
```

The AudioScrobbler Data Set

AudioScrobbler was one of the first recommender platforms developed for online music portals such as last.fm (similar to Spotify, Amazon Music, iTunes, etc.)

- Obviously, the goal is to recommend new artists to users who have an existing profile of artists they already played.
- Available at: https://datahub.io/dataset/audioscrobbler (and on Moodle!)
- The archive contains 3 files of approximately 135 MB in compressed format.

user_artist_data.txt

- "Implicit" collection of 24 Mio individual user ratings (141K users, 1.6M artists):
- About 426 MB in uncompressed space-separated format:

```
<user_id <SPACE> artist_id <SPACE> number_of_times_played>
```

artist_data.txt

- Mappings of about 1.8 Mio artist ids to their actual names in TSV format:

```
<artist_id <TAB> artist_name>
```

artist_alias.txt

- Pairwise artist aliases, i.e., capturing different name variations of a same artist:

```
<artist_id <SPACE> artist_alias_id>
```

Outline of Chapter 3

3.1 Matrix Factorization

- Approximate Matrix Factorization
- Alternating Least Squares Algorithm
- Performance Comparison

3.2 Recommender Systems in Spark's MLlib

- Loading and Exploring the Data Set
- Broadcasting Closure Variables
- Preparing the Training Data
- Evaluation of Recommendation Quality
- Hyper-Parameter Tuning

3.1 Matrix Factorization

Matrix Factorization: Goal & Examples

Goal: decompose a non-negative $n \times m$ matrix $\boldsymbol{R}$ into two smaller matrices $\boldsymbol{X}^T$ and $\boldsymbol{Y}$ of sizes $n \times k$ and $k \times m$, respectively, such that $\boldsymbol{X}^T \times \boldsymbol{Y}$ "*best resembles*" $\boldsymbol{R}$, i.e.:

$$\boldsymbol{R} \approx \boldsymbol{X}^T \times \boldsymbol{Y}$$

- $\boldsymbol{R}$ is typically *sparse* but of *high rank*.
- $\boldsymbol{X}$ and $\boldsymbol{Y}$ are typically *dense* and also of *high rank*.
- Parameter k controls the amount of "**latent features**". These can be thought of as the different types of music tastes represented in the AudioScrobbler data set.

Examples: (using $k = 2$)

Low-rank matrix $\boldsymbol{R}$:

- $$\boldsymbol{R} = \begin{bmatrix} 1 & 2 & 3 & 5 \\ 2 & 4 & 8 & 12 \\ 3 & 6 & 7 & 13 \end{bmatrix} = \underbrace{\begin{bmatrix} 0.76 & 1.05 \\ 2.55 & 2.03 \\ 1.28 & 3.23 \end{bmatrix}}_{\boldsymbol{X}^T} \times \underbrace{\begin{bmatrix} 0.06 & 0.24 & 2.03 & 2.15 \\ 0.90 & 1.72 & 1.37 & 3.17 \end{bmatrix}}_{\boldsymbol{Y}} = \begin{bmatrix} 1.00 & 2.00 & 2.99 & 4.99 \\ 2.00 & 4.12 & 7.96 & 11.94 \\ 2.99 & 5.87 & 7.01 & 13.01 \end{bmatrix}$$

High-rank matrix $\boldsymbol{R}$:

- $$\boldsymbol{R} = \begin{bmatrix} 5 & 8 & 0 & 0 \\ 0 & 0 & 9 & 8 \\ 6 & 0 & 0 & 0 \end{bmatrix} \approx \underbrace{\begin{bmatrix} 1.62 & 1.92 \\ 3.06 & 1.47 \\ 2.12 & 2.07 \end{bmatrix}}_{\boldsymbol{X}^T} \times \underbrace{\begin{bmatrix} 1.52 & 2.01 & 2.46 & 2.06 \\ 1.33 & 2.46 & 0.96 & 1.11 \end{bmatrix}}_{\boldsymbol{Y}} = \begin{bmatrix} 5.01 & 7.98 & 6.47 & 5.81 \\ 6.75 & 10.96 & 8.99 & 7.99 \\ 5.98 & 9.17 & 7.26 & 6.65 \end{bmatrix}$$

See: https://www.youtube.com/watch?v=o8PiWO8C3zs for solving the low-rank example.

Alternating Least Squares (ALS): Objective Function

Solving a set of linear equations even for the low-rank case is quite cumbersome.

- If k is less than the rank of matrix $\boldsymbol{R}$, then there is no exact solution at all, i.e., the resulting factorization is going to be "lossy".
- The resulting approximation of $\boldsymbol{R}$ however has the desired effect that $r_{ui} \approx x_u^T y_i$ can provide also *new recommendations* for user u with respect to artist i.

$$X = \begin{bmatrix} | & & | \\ x_1 & \cdots & x_n \\ | & & | \end{bmatrix}, Y = \begin{bmatrix} | & & | \\ y_1 & \cdots & y_m \\ | & & | \end{bmatrix}$$

Idea:

- *Fix* the **two hyper-parameters** k (e.g., 10 to 100 limiting X, Y) and λ (e.g., 0 to 1)
- *Minimize*, with some degree of freedom, the following **objective function**:

$$\min_{X,Y} \sum_{r_{ui}\ observed} (r_{ui} - x_u^\intercal y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right) \quad (1)$$

Approximation of $\boldsymbol{R} \approx \boldsymbol{X}^T \times \boldsymbol{Y}$

"Regularization" with parameter λ

Iterative ALS Algorithm (Single Machine)

Algorithm 1 ALS for Matrix Completion

Initialize X, Y

repeat

for $u = 1 \dots n$ **do**

$$x_u = \Big(\sum_{r_{ui} \in r_{u*}} y_i y_i^\intercal + \lambda I_k \Big)^{-1} \sum_{r_{ui} \in r_{u*}} r_{ui} y_i \tag{2}$$

end for

for $i = 1 \dots m$ **do**

$$y_i = \Big(\sum_{r_{ui} \in r_{*i}} x_u x_u^\intercal + \lambda I_k \Big)^{-1} \sum_{r_{ui} \in r_{*i}} r_{ui} x_u \tag{3}$$

end for

until convergence

(repeated either until convergence or for a fixed number of iterations)

- Update costs of each x_u: $O(n_u k^2 + k^3)$ where user u has listened to n_u artists
- Update costs of each y_i: $O(n_i k^2 + k^3)$ where artist i has been listened to by n_i users

Distributed ALS: Method 1

By using **fully distributed** encodings of the three matrices as RDDs:

$$\boldsymbol{R}: RDD\big((u, i, r_{ui}), \ldots\big)$$
$$\boldsymbol{X}: RDD(x_1, \ldots, x_n)$$
$$\boldsymbol{Y}: RDD(y_1, \ldots, y_m)$$

To compute Eq. (2) (see previous slide)

- Join $\boldsymbol{R}$ with $\boldsymbol{Y}$ on i (the items/artists to be recommended)
- Map each y_i into $y_i y_i^T$ and change the key to u (the users)
- ReduceByKey on u to compute $\sum_i y_i y_i^T$ and invert the resulting matrix
- ReduceByKey on u to compute $\sum_i r_{ui} y_i$
- Join & Map to multiply the latter two RDDs

Perform a similar series of transformations also to compute Eq. (3)

Repeat until convergence.

Distributed ALS: Method 2

By using one RDD for $\boldsymbol{R}$ and by **broadcasting** the local matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$:

$$\boldsymbol{R}\text{: } RDD\big((u, i, r_{ui}), \dots\big)$$

Partition $\boldsymbol{R}$ into

- $\boldsymbol{R}_{1,1}, \dots, \boldsymbol{R}_{1,p}$ by same users u and
- $\boldsymbol{R}_{2,1}, \dots, \boldsymbol{R}_{2,p}$ by same items/artists i

such that each partition p is assigned to one worker node.

Repeat

- Broadcast $\boldsymbol{X}$ and $\boldsymbol{Y}$ among all worker nodes
- MapPartitions on $\boldsymbol{R}_{1,1}, \dots, \boldsymbol{R}_{1,p}$ and $\boldsymbol{Y}$ to compute Eq. (2)
- MapPartitions on $\boldsymbol{R}_{2,1}, \dots, \boldsymbol{R}_{2,p}$ and $\boldsymbol{X}$ to compute Eq. (3)

until convergence.

(Caution: broadcasting $\boldsymbol{X}, \boldsymbol{Y}$ is tricky in Spark because broadcast variables are immutable!)

Distributed ALS Algorithm: Runtime Comparison

System	Wall-clock /me (seconds)
MATLAB	15,443
Mahout	4,206
GraphLab	291
MLlib	481

- Dataset: scaled version of Netflix data (9X in size!)
- Cluster: 9 machines
- MLlib is an order of magnitude faster than Mahout (based on Hadoop)
- MLlib is within factor of 2 of GraphLab

3.2 Recommender Systems in Spark's MLlib

Load the Data Set

- Copy artist_alias.txt, artist_data.txt, and user_artist_data.txt into either your local directory or your home directory on IRIS. We will assume you are using your local home directory for the following steps.
- To run ALS from your Scala implementation, specify --driver-memory 4g to make sure Spark reserves enough main memory to store your repeatedly read RDD's from cache.

  spark-shell --driver-memory 4g

- Start by reading the three data files into one flat RDD each:

```
val rawArtistAlias = sc.textFile("./artist_alias.txt")
val rawArtistData = sc.textFile("./artist_data.txt")
val rawUserArtistData = sc.textFile("./user_artist_data.txt")
```

- The latter file user_artist_data.txt is by far the largest file with 426 MB.

Explore the Data Set: Get Some Basic Statistics

- Spark's ALS implementation uses 32-bit integers to represent row and column indices. That is, no entries in user_artist_data.txt may be larger than `Integer.MAX_VALUE`, which is $2^{32-1}-1 = 2147483647$ for a signed integer.
- First, explore the range of values in user_artist_data.txt:

```
rawUserArtistData.map(_.split(' ')(0).toDouble).stats()
rawUserArtistData.map(_.split(' ')(1).toDouble).stats()
```

- Here, `split()` splits each line by the provided truncation character(s).
- `stats()` provides a concise summary about the distribution of the values within the provided list (including the maximum and minimum value).

Prepare the Data Set: Parsing & Error Handling

- Prepare a PairRDD, called `artistByID`, that holds the mappings from artist ids (Int) to their actual names (String):

```
val artistByID = rawArtistData.map { line =>
  val (id, name) = line.span(_ != '\t') (id.toInt, name.trim) }
```

- Here, `span()` attempts to split each line by its first tab. It then parses the first portion as the numeric artist id and retains the rest as the name.

- However, a small number of the lines are corrupt, such that either no id or no name can be parsed. So here is an improved version:

```
val artistByID = rawArtistData.flatMap {
  line => val (id, name) = line.span(_ != '\t')
  if (name.isEmpty) { None } else {
    try {
      Some((id.toInt, name.trim)) }
    catch { case e: NumberFormatException => None }
} }
```

Prepare the Data Set: Resolving Aliases

- The artist_alias.txt file contains two ids per line, again separated by a tab. This file is relatively small, only containing about 200,000 entries.
- Both ids are indeed used in the artist_data.txt and user_artist_data.txt file, but they both represent the same real-world artist.
- Hence, it will be useful to collect these aliases into a local map, called `artistAlias`, thus mapping "bad" artist ids to "good" ones.
- Again, some lines are missing the first id, and thus are skipped:

```
val artistAlias = rawArtistAlias.flatMap { line =>
  val tokens = line.split('\t')
  if (tokens(0).isEmpty) { None } else {
    Some((tokens(0).toInt, tokens(1).toInt))
  } }.collectAsMap()
```

- Also check out some of these actual aliases:

```
artistByID.lookup(6803336).head
artistByID.lookup(1000010).head
```

Broadcasting Closure Variables

- Recall that the **closure of a function** contains all the data structures within the current scope of the runtime environment that are referenced by the function's body.
- That is, running a distributed computation with a map function that references additional data structures requires Spark to **serialize the entire function's closure into a binary format** and then **repeatedly ship** this closure **for every task** that is processed by the function (e.g., one line of artist_user_data.txt).
- This is exactly the use-case for a **broadcast variable** (see Chapter 1.2):

```
val bArtistAlias = sc.broadcast(artistAlias)
```

- The broadcast variable `bArtistAlias` is **shipped only once for every executor** of a map function (e.g., based on the 6 to 7 file splits we expect to obtain for artist_user_data.txt).

Prepare & Cache the Training Data

- Our final preparation of the training data consists of

1. mapping all artist ids to their canonical ids (the second column from artist_alias.txt), and
2. transforming `rawUserArtistData` into a `Rating` data structure, which is the default data structure used by Spark to hold user-artist-rating triples.

```
import org.apache.spark.mllib.recommendation._

val trainData = rawUserArtistData.map {
  line => val Array(userID, artistID, count) =
    line.split(' ').map(_.toInt)
  val finalArtistID = bArtistAlias.value.getOrElse(artistID,
    artistID)
  Rating(userID, finalArtistID, count)
}.cache()
```

- The training data is used many times by the ALS algorithm, and hence it should be cached!

Train a First Recommender Model

- Trigger the ALS computation in Spark using the `trainData` RDD:

```
val model = ALS.trainImplicit(trainData, 10, 5, 0.01, 1.0)
```

- Note:
  - Running ALS may take a couple of minutes when using the entire AudioScrobbler training set on the IRIS HPC cluster!
  - Thus, always use a smaller training set (e.g., use `.sample(false, 0.01)`) to first debug your algorithms!

- Inspect the resulting model:

```
model.userFeatures.mapValues(_.mkString(", ")).first()
```

(which returns nothing else but the first row of the $\boldsymbol{X}$ matrix in the factorization)

See https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html for a more comprehensive API documentation of the `org.apache.spark.mllib.recommendation` package.

Spot-Checking Users & Recommendations (I)

- Verify the viability of your model by comparing a user's actually played artists with the recommendations we would obtain from the model:

```
val rawArtistsForUser = rawUserArtistData.map(_.split(' ')).
  filter { case Array(user,_,_) => user.toInt == 2093760 }
val existingArtists =
  rawArtistsForUser.map {case Array(_,artist,_) => artist.toInt}
    .collect().toSet
artistByID.filter {case (id, name) =>
  existingArtists.contains(id)}.values.collect().foreach(println)
```

- And compare these to the top-5 recommended artists for this user:

```
val recommendations = model.recommendProducts(2093760, 5)
recommendations.foreach(println)
```

Note: the new ratings exclude artists that the user actually played already!

Spot-Checking Users & Recommendations (II)

- Finally, inspect the top-5 recommendations by looking up the artists' names in the `artistByID` map.

```
val recommendedArtistIDs = recommendations.map(_.product).toSet
artistByID.filter { case (id, name) =>
   recommendedArtistIDs.contains(id)
}.values.collect().foreach(println)
```

After this step, you may want to check the progress and current memory consumption of your running Spark jobs: http://localhost:4040

- Enable port forwarding from an IRIS node to your localhost:
  - ssh -p 8022 <student-id>@access-iris.uni.lu
  - srun -p batch --time=00:30:0 -N 2 -c 12 --pty bash –i (remember the node id!)
  - ./launch-spark-shell.sh
- Then open a second terminal and replace XXX with the id of your IRIS node:
  - ssh -p 8022 -L 4040:iris-XXX.iris-cluster.uni.lux:4040 <student-id>@access-iris.uni.lu

Evaluating Recommendation Quality: ROC & AUC

Receiver Operating Characteristic (ROC)

- Suppose we process a ranked list of results from top to end.
- At each rank *r*, we measure the **true positive rate** (the number of correctly returned items at rank *r*) and the **false positive rate** (the number of erroneously returned items at rank *r*).
- The function obtained by plotting these points over all ranks is called the **receiver operator characteristic**.

Area Under the Curve (AUC)

- The **area under the ROC curve** is a typical measure for the goodness of a recommender system. It is high (i.e., approaching 1) if a system returns more correct items than erroneous items. It is 0.5 if results are random (correct and erroneous items are returned interchangeably at the ranks).
- Since ROC is not a smooth (i.e., differentiable) function, we usually approximate it by **averaging over various ROC points**.

Receiver-Operator Statistic (ROC)

The so-called "**Receiver-Operator Characteristics**" (ROC) plots the true-positive rate (TPR) against the false-positive rate (FPR) of a system.

TPR (i.e., the relative size of TP at a given rank) is also referred to as $sensitivity$, which is identical to $recall$, and describes how good a system is at returning the relevant items.

FPR (i.e., the relative size of FP at a given rank) is also referred to as $1 - specificity$, where $specificity$ describes how good a system is at avoiding the non-relevant items.

- Formally, we have:

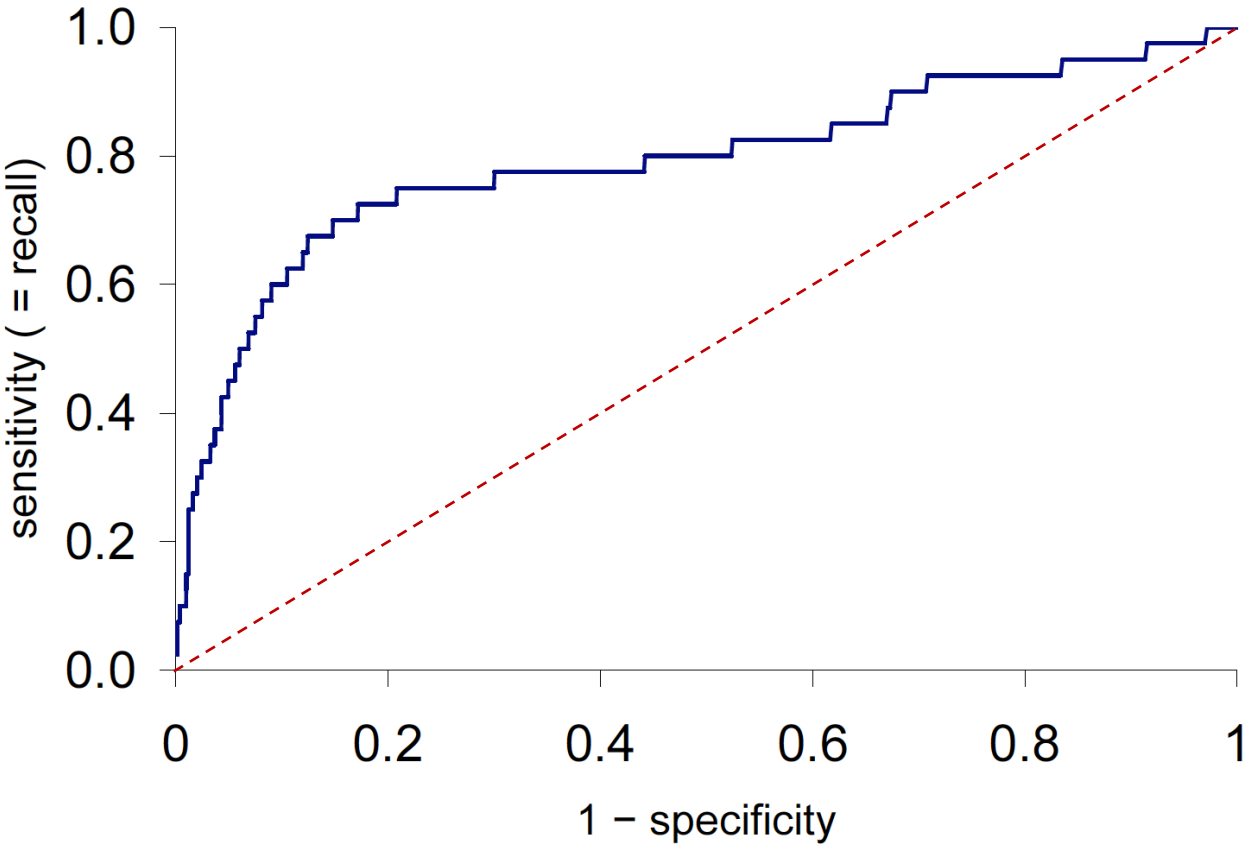
$$\mathrm{TPR} = sensitivity = \frac{|TP|}{|TP|+|FN|}$$

$$\mathrm{FPR} = 1 - specificity = 1 - \frac{|TN|}{|FP|+|TN|} = \frac{|FP|}{|FP|+|TN|}$$

The **Area-Under-the-Curve** (AUC) measure then is defined as the average TPR value over a number of predefined FPR points.

See also: https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Illustration of ROC



- An example of a typical ROC curve is shown by the upper blue plot.
- Note that a "random" system, which interchangeably retrieves a relevant and then a non-relevant result, results in a straight line from (0,0) to (1,1).

ROC & AUC Example Calculation

Create new RDD: `predictionsAndLabels`

Rank	Predicted Relevance (ranked)	Actual Relevance (binary)	TPR = $\frac{\lvert TP \rvert}{\lvert TP \rvert + \lvert FN \rvert}$	FPR = $\frac{\lvert FP \rvert}{\lvert FP \rvert + \lvert TN \rvert}$
1	0.43	1	1/(1+6) = 0.14	0/(0+3) = 0.00
2	0.34	1	2/(2+5) = 0.28	0/(0+3) = 0.00
3	0.21	0	2/(2+5) = 0.28	1/(1+2) = 0.33
4	0.17	0	2/(2+5) = 0.28	2/(2+1) = 0.67
5	0.12	1	3/(3+4) = 0.42	2/(2+1) = 0.67
6	0.11	1	4/(4+3) = 0.57	2/(2+1) = 0.67
7	0.08	1	5/(5+2) = 0.71	2/(2+1) = 0.67
8	0.06	1	6/(6+1) = 0.86	2/(2+1) = 0.67
9	0.00	0	6/(6+1) = 0.86	3/(3+0) = 1.00
10	-0.01	1	7/(7+0) = 1.00	3/(3+0) = 1.00

AUC = 0.54

Fortunately, the ROC and AUC measures are readily built into Spark's `BinaryClassificationMetrics` package:
https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html

Evaluating Recommendation Quality: Cross-Validation

Cross-Validation

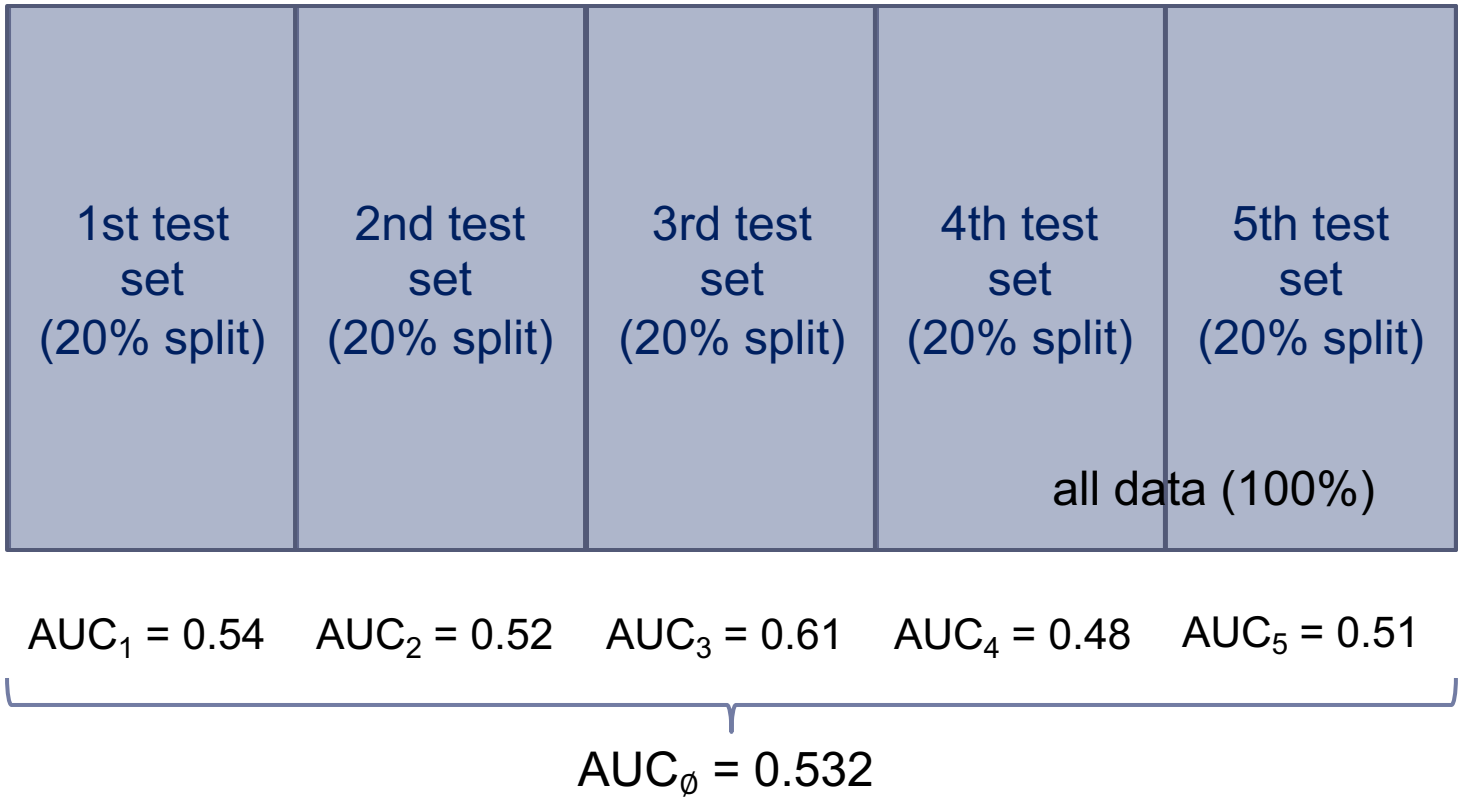
- Withhold a (typically small) fraction (e.g., 90%) of the overall data set for training, use the remaining part (e.g., 10%) for testing the accuracy of the learned model.

K-fold Cross-Validation

- Iterate the cross-validation step *k* times by splitting the overall data set into *k* disjoint partitions.
- Use *k*-1 partitions for training and the remaining partition for testing at each iteration, then subsequently shift the training and testing partitions.
- Finally measure the accuracy or AUC at each split and use the averaged result over the *k* steps to select the best hyper-parameter setting.

K-Fold Cross-Validation Illustration

5-fold cross-validation: 5x split each using 80% for training data and 20% for testing



Fortunately, also cross-validations are readily built into Spark's `CrossValidator` package:

https://spark.apache.org/docs/latest/ml-tuning.html

Baseline Selection

- Similarly to our earlier example about Decision Trees, we may also want to compare our new recommender model to a **reasonable baseline**.
- So how good is a system that would simply recommend the same "*most frequently listened to*" artists to every user?

```
val artistsTotalCount = trainData.map(r => (r.product,
  r.rating)).reduceByKey(_ + _).collect().sortBy(-_._2)
def predictMostPopular(user: Int, numArtists: Int) = {
  val topArtists = artistsTotalCount.take(numArtists)
  topArtists.map{case (artist, rating) => Rating(user,
     artist, rating)}
}
val auc = ... // see RunRecommender-shell.sh on Moodle!
```

- Note: `predictMostPopular(...)` thus creates artificial `Rating` objects based on the total number of times all users listened to the artists!
- Here, `predictMostListened` even yields an AUC value of about 0.930!

Hyper-Parameters of Spark's `MatrixFactorizationModel`

So far, the so-called **hyper-parameters** used to build the `MatrixFactorization-Model` were simply given. They are not learned by the algorithm and must be chosen by the caller. The following arguments are used by the function:

```
ALS.trainImplicit(trainData, rank, iterations, lambda, alpha)
```

- `rank = 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.
- `iterations = 5`
  The number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.
- `lambda = 0.01`
  A standard regularization parameter. Higher values resist overfitting, but values that are too high hurt the factorization's accuracy.
- `alpha = 1.0`
  Controls the relative weight of observed versus unobserved user-product interactions in the factorization (internal Spark parameter).

Hyper-Parameter Tuning

- Using the idea of cross-validation, we now have a powerful approach to systematically investigate the **hyper-parameter space**:

```
val evaluations = for(rank <- Array(10, 50);
  lambda <- Array(1.0, 0.0001);
  alpha <- Array(1.0, 40.0))
yield {
 val model = ALS.trainImplicit(trainData, rank, 10, lambda, alpha)
 val auc = ... // see RunRecommender-shell.sh on Moodle!
 ((rank, lambda, alpha), auc)
}
evaluations.sortBy(_._2).reverse.foreach(println)
```

- Here, we set `rank` to `{10, 50}`, `lambda` to `{1.0, 0.0001}` and `alpha` to `{1, 40}`, hence we obtain 8 combinations of parameters.
- The resulting combination of `rank=10`, `lambda=1.0` and `alpha=40.0` yields an average AUC value of 0.976 over k = 10 cross-validations.

Creating Batch Recommendations

- Spark's ALS implementation only allows one recommendation at a time.
- However, we can still slightly improve the performance for making recommendations to multiple users by **batch-processing** them from within a preselected list of users:

```
val someUsers = trainData.map(x => x.user).distinct().take(10)
val someRecommendations = someUsers.map(userID =>
  model.recommendProducts(userID, 5))
someRecommendations.map(recs => recs.head.user + " -> " +
  recs.map(_.product).mkString(", ")).foreach(println)
```

Extensions

- **Fix Data Errors**
  - Improve exception handling for more cases of errors: can some lines still be saved?
- **Remove Obvious Outliers**
  - Some users may have played suspiciously many artists; while some artists are played by suspiciously many users.
  - Often, removing the most frequent and/or infrequent patterns from the data set may help to improve the prediction accuracy.
- **More Tuning**
  - Systematically explore a larger parameter space.
  - Use ensemble techniques, e.g., both collaborative and content-based filtering.

→ Compare also to the "Data Science Workflow" slide from Chapter 2!