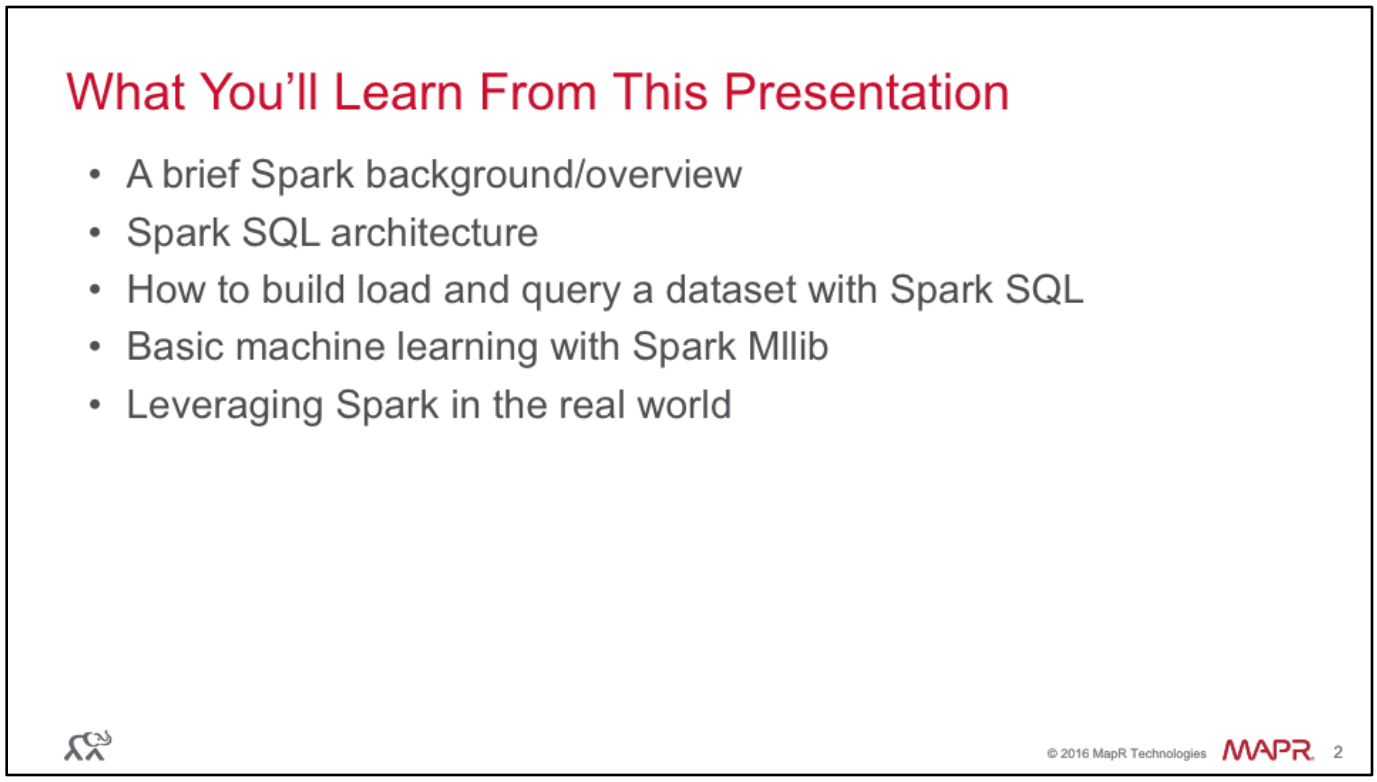
Spark SQL & Machine Learning
A Practical Demonstration

Craig Warman



- Spark Background/Overview
  - Brief Spark background
  - Tthe Spark+Hadoop team
  - Spark's five main components
- Spark SQL Architecture
  - Features, Languages, How DataFrames work, The SQLContext, Data sources
- Loading And Querying a Dataset with Spark SQL
  - Live demonstration of setting up a SQLContext
  - Loading it with data
  - Running queries against it
- Machine Learning with Spark MLlib
  - Collaborative filtering basics
  - Alternating Least Squares (ALS) algorithm
  - Live demo of a simple recommender model and training-test loop iterations

- How to connect to Spark SQL using ODBC/JDBC
  - Live demonstration of how to leverage Spark SQL ODBC/JDBC connectivity using Tableau
- Next steps
  - Some Real-World Use Cases
  - Basically answer the questions "What's it good for?" and "Who's using this?"
  - How to download a ready-to-use sandbox VM

Spark Background / Overview

- A powerful open source processing engine
- Promoted to Apache top-level project in 2014
- Key Drivers:
  - Rich and well-documented APIs
  - Supports multiple languages
  - Optimized to run in-memory
- It is not a Hadoop replacement
  - Though it may eventually replace MapReduce

 MAPR 3

Spark began life in 2009 as a project within the AMPLab at the University of California, Berkeley.

Spark became an incubated project of the Apache Software Foundation in 2013
Was promoted to Foundation top-level project in early 2014

- Is currently one of the most active projects managed by the Foundation

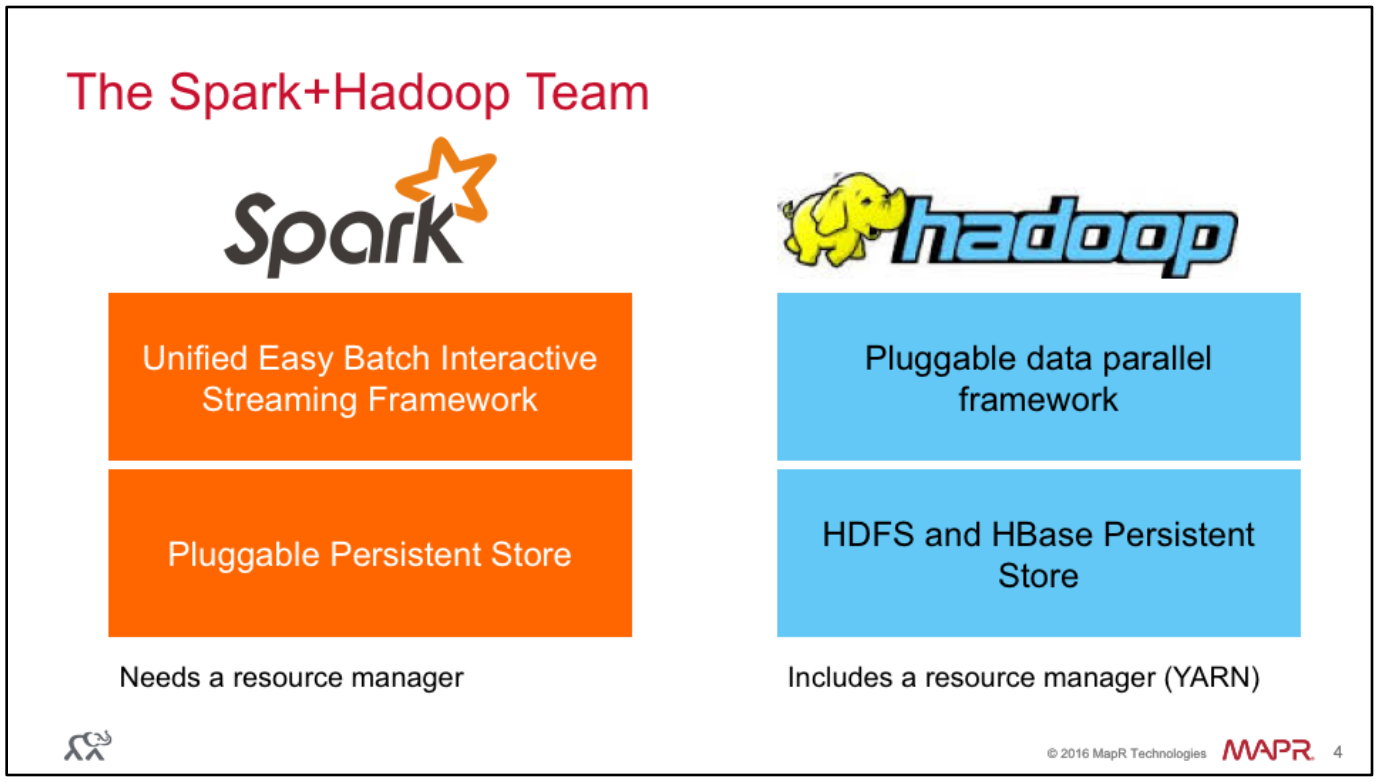
Key drivers (Reasons why you might select Spark for a data project):

- Rich and well-documented API designed specifically for interacting with data at scale.
- Supports Java, Scala, Python, R, and SQL
- Spark was optimized to run in memory from the beginning
  - Helps it process data far more quickly than

alternative approaches like Hadoop's MapReduce, which tends to write data to and from computer hard drives between each stage of processing.

Spark is not, despite the hype, a replacement for Hadoop. Nor is MapReduce dead.

- Hadoop is a platform that encompasses a wide variety of technologies
- Spark is faster than MapReduce for iterative algorithms that fit data in memory.
    - So, even though you can run Spark in stand-alone mode, doing so means you completely miss out on Hadoop's ability to run multiple types of workloads (incl. advanced analytics with Spark) on the same data at the same time.
    - In other words, Spark without Hadoop is just another silo.



Important to remember: Hadoop is more than just MapReduce

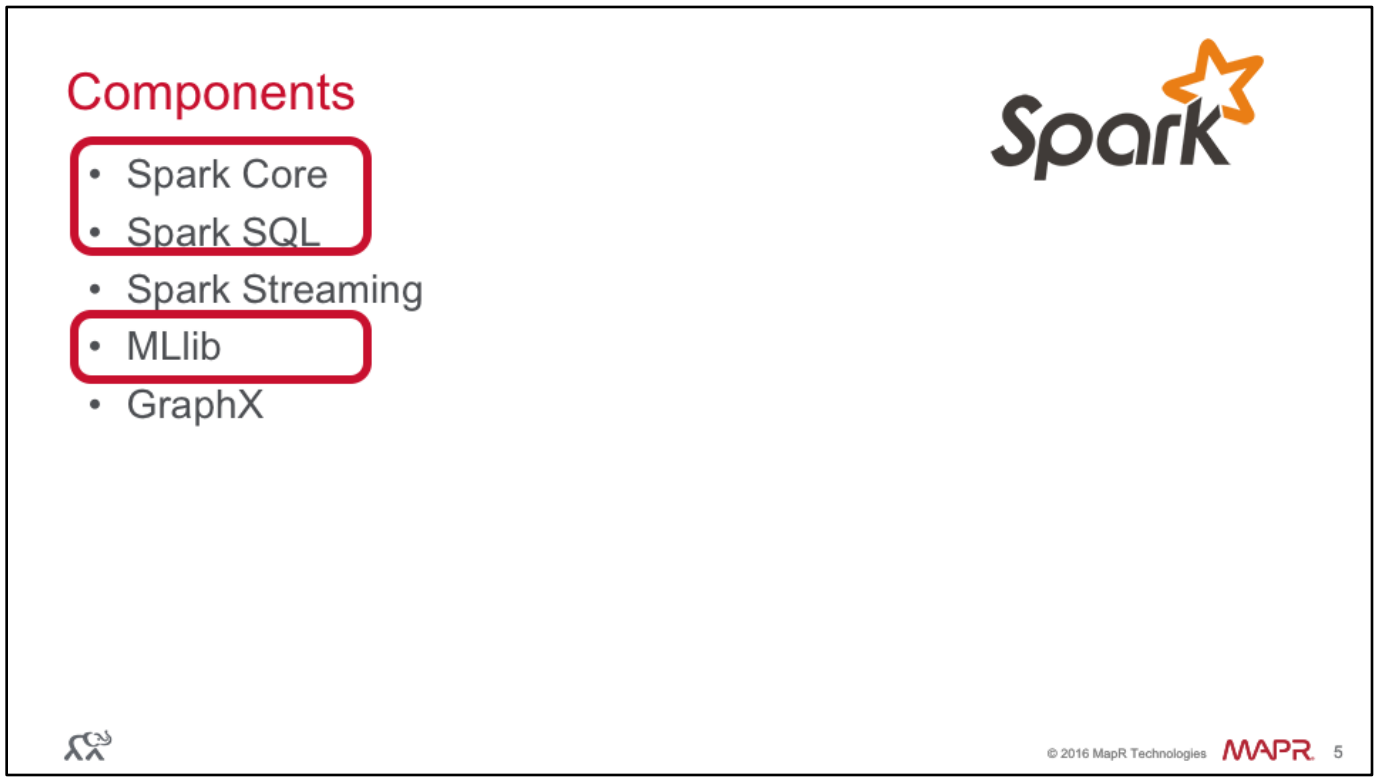
Spark and MapReduce are:

- Scalable frameworks for executing custom code on a cluster
- Nodes in the cluster work independently to process fragments of data and also combine those fragments together when appropriate to yield a final result
- Can tolerate loss of a node during a computation
- **Require a distributed storage layer for common data view**

Spark is often deployed in conjunction with a Hadoop cluster

- On its own, Spark isn't well-suited to production workloads
- It needs a resource manager
    - YARN, for instance, which takes responsibility for scheduling tasks across cluster nodes
    - A distributed filesystem

- Data security



Apache Spark consists of Spark Core and a set of libraries.

Spark Core

- This is the distributed execution engine
- Implements a programming abstraction known as Resilient Distributed Datasets RDDs
- Provides APIs for S=Java, Scala, and Python
- Some documentation shows R as part of the core, others show it as a separate module. The R module was added to the 1.4 release.

Spark SQL

- This is the module for working with structured data using SQL
- Supports the open source Hive project, along with ODBC and JDBC

Spark Streaming

- Enables scalable and fault-tolerant processing of data streams
- Supports sources like Flume (for data logs) and Kafka (for messaging)

MLlib

- Scalable machine learning library
- Implements commonly-used machine learning and statistical algorithms
- These algorithms are often iterative
    - Spark's ability to cache the dataset in memory greatly speeds up such iterative data processing
    - So it's an ideal processing engine for implementing such algorithms

GraphX

- Supports analysis of and computation over graphs of data
- Includes a number of widely understood graph algorithms, including PageRank.
- Graph databases are well-suited for analyzing interconnections
- They're useful for working with data that involve complex relationships (such as social media)

This presentation will focus primarily on the Spark Core API using Scala, along with SQL and the machine learning library

Resilient Distributed Datasets (RDDs)

- Core programming abstraction
- Supports in-memory data storage
- Fault-tolerant and efficient
- Immutable (they cannot be changed)

RDDs are a core programming abstraction at the heart of Spark.

Allows cross-cluster distribution

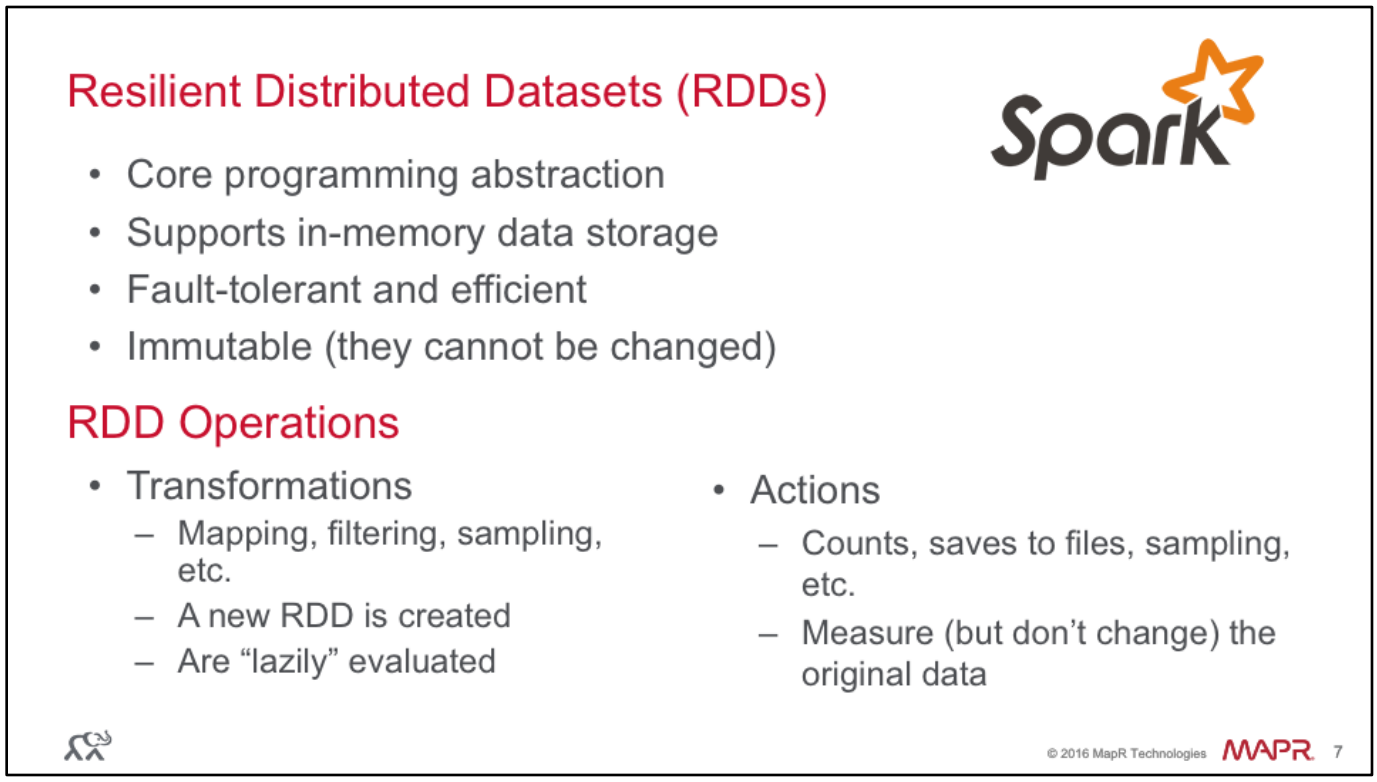
Fault-tolerance is achieved by tracking the lineage of transformations applied to coarse-grained sets of data.

Efficiency is achieved by parallelization of processing across multiple cluster nodes, and by minimization of data replication between those nodes.

RDDs remain in memory to enhance performance
Particularly true in use cases with a requirement for iterative queries or processes.
We will observe this in the demo

This immutability is important.

The chain of transformations from RDD1 to RDDn are logged, and can be repeated in the event of data loss or the failure of a cluster node.



There are two basic types of operations

Transformations essentially change the data, so a new RDD is created because the original cannot be changed
They're lazily evaluated – meaning that they aren't executed until a subsequent action has a need for a result
This improves performance because it can avoid unnecessary processing

Actions measure – but don't change – the original data.
They essentially force processing to take place

Data Frames

- Distributed collection of data
- Organized into named columns
- Basically similar to a table
- APIs for Python, Java, Scala, and R

The DataFrames API was added to Spark in 2015

If you're an R or Python programmer then you'll be familiar with the concept here

They have the ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster

Basically this is structured data, just like a table in a relational database

Which leads us to…

Data Frames

- Distributed collection of data
- Organized into named columns
- Basically similar to a table
- APIs for Python, Java, Scala, and R

Spark SQL

- Enables JDBC/ODBC access from BI tools
- Supports Hive, Avro, Parquet, ORC, JSON, and JDBC sources
- Supported by Python, Java, Scala, and R APIs

…Spark SQL

Basically this gives Spark programs the ability to query structured data using SQL

It also enables accessibility from BI tools such as Tableau
In fact we'll explore this later in the demo

You've got a wide variety of supported data sources
Joins between dataframes are supported as well

So this can be a very powerful capability

Demo #1: RDDs, Data Frames, and SQL

My Environment

- 13” Mac Laptop
- 16GB RAM, 256GB SDD
- 2.9 GHz Intel Core i5 (2 cores)
- Spark installed using Homebrew

```
brew install apache-spark
```

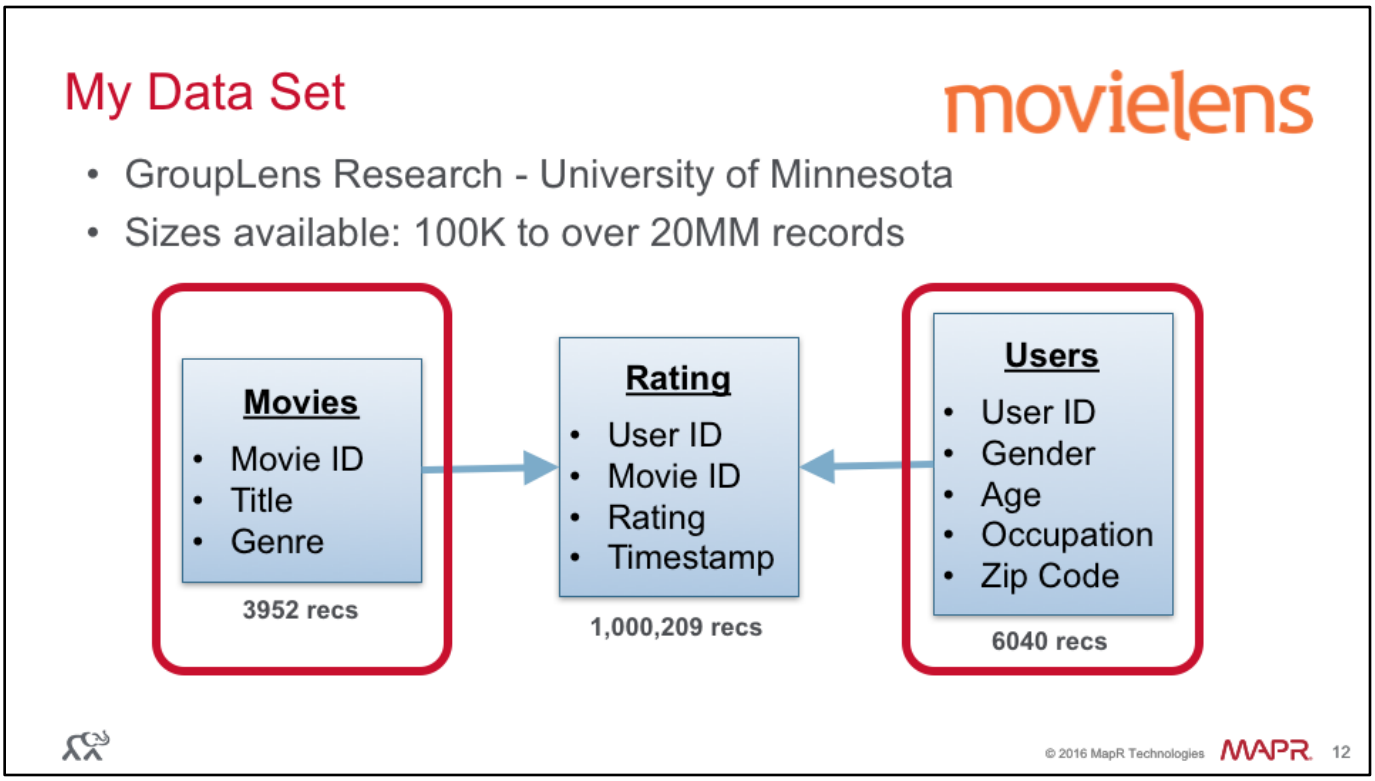
Scala Shell: `spark-shell`

Python Shell: `pyspark`

SQL Shell: `spark-sql`

So this is not a fast machine I’m working with

Installation using Homebrew is dead simple.
Basically Homebrew downloads Apache Spark



MovieLens is maintained by members of GroupLens Research at the University of Minnesota

I'm using one of the smaller sets, collected in 2000

We'll focus on the Users and Movies files for this first demo

Source: http://grouplens.org/datasets/movielens/

Define Schemas

```
case class Movie
   (movieId: Int,
    title: String,
    genres: Seq[String])
```

```
case class User
   (userId: Int,
    gender: String,
    age: Int,
    occupation: Int,
    zip: String)
```

 MAPR 13

These are Scala case classes that define schemas corresponding to their respective files

Define Parsing Functions

```
def parseMovie(str: String): Movie = {
      val fields = str.split("::")
      assert(fields.size == 3)
      Movie(fields(0).toInt, fields(1).toString, Seq(fields(2)))
}

def parseUser(str: String): User = {
      val fields = str.split("::")
      assert(fields.size == 5)
      User(fields(0).toInt, fields(1).toString,
           fields(2).toInt, fields(3).toInt, fields(4).toString)
}
```

These functions parse lines from each file into the corresponding case classes defined

Note the references to the earlier-defined classes

Load Files into RDDs

```
val moviesRDD = sc.textFile("movies.dat").map(parseMovie)
val usersRDD = sc.textFile("users.dat").map(parseUser)
```

These functions parse lines from each file into the corresponding case classes defined

Note the **map** method – this invokes each of the parsing functions defined earlier

Load Files into RDDs

```
val moviesRDD = sc.textFile("movies.dat").map(parseMovie)
val usersRDD = sc.textFile("users.dat").map(parseUser)
```

Convert RDDs into Data Frames

```
val moviesDF = moviesRDD.toDF()
val usersDF = usersRDD.toDF()
```

 16

The **toDF** method makes the conversion happen

Examine the schemas once loaded

Load Files into RDDs

```
val moviesRDD = sc.textFile("movies.dat").map(parseMovie)
val usersRDD = sc.textFile("users.dat").map(parseUser)
```

Convert RDDs into Data Frames

```
val moviesDF = moviesRDD.toDF()
val usersDF = usersRDD.toDF()
```

Register Data Frames

```
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")
```

registerTempTable() creates an in-memory table that is scoped to the cluster in which it was created.
The data is stored using Hive's highly-optimized, in-memory columnar format.
This allows it to be queried with SQL statements that are executed using methods provided by **sqlContext**

We'll look at a **more permanent method** for materializing data frames at the end of this presentation.

I referenced this document:
https://forums.databricks.com/questions/400/what-is-the-difference-between-registertemptable-a.html

Query #1 – What are some of the movies?

```
sqlContext.sql("SELECT * FROM movies").show()
```

Just a simple query here

Query #1 – What are some of the movies?

```
sqlContext.sql("SELECT * FROM movies").show()
```

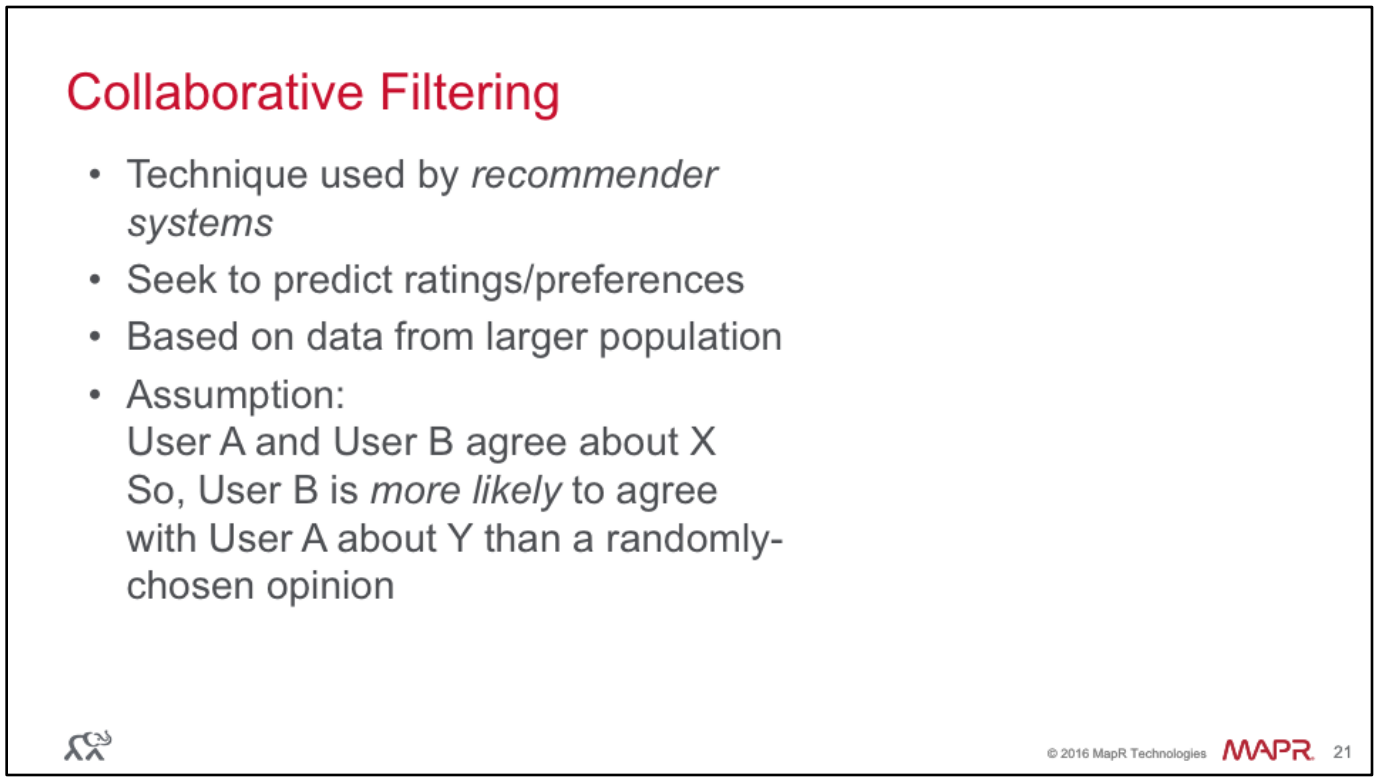
Query #2 – What are some user demographics?

```
sqlContext.sql("
   SELECT
      age,
      sum(if(gender='M',user_count,0)) AS M,
      sum(if(gender='F',user_count,0)) AS F
   FROM
      (SELECT age, gender, count(*) AS user_count
       FROM users GROUP BY age, gender) AS ud
   GROUP BY age ORDER BY age").show()
```

A little more involved query that utilizes a nested view

So we've covered the basics - RDDs, Data Frames, and SQL

Let's move on to the more interesting stuff



Collaborative filtering (CF) is a technique used by some recommender systems.

Recommender systems essentially seek to predict the 'rating' or 'preference' that a user would give to an item.

So, **Collaborative Filtering** tries to make these predictions (we call this “filtering”) about the interests of a user by collecting information from many users (hence “collaborating”)

Collaborative Filtering

- Technique used by *recommender systems*
- Seek to predict ratings/preferences
- Based on data from larger population
- Assumption:
  User A and User B agree about X
  So, User B is *more likely* to agree with User A about Y than a randomly-chosen opinion

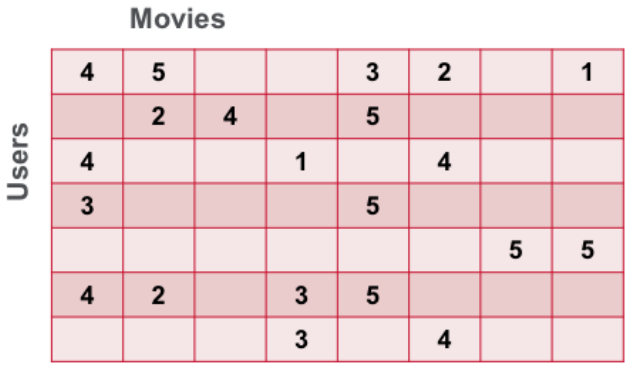
I borrowed this animated GIF from Wikipedia
Source: https://en.wikipedia.org/wiki/Collaborative_filtering

This is an example of collaborative filtering.
At first, people rate different items (like videos, images, games).
Then, the system makes predictions about a user's rating for an item not rated yet.
The new predictions are built upon the existing ratings of other users with similar ratings with the active user.

In the image, the system predicts that the user will not like the video.

Alternating Least Squares (ALS) Algorithm

- Overall goal: Complete the rating matrix
- The rating matrix is your central data structure.



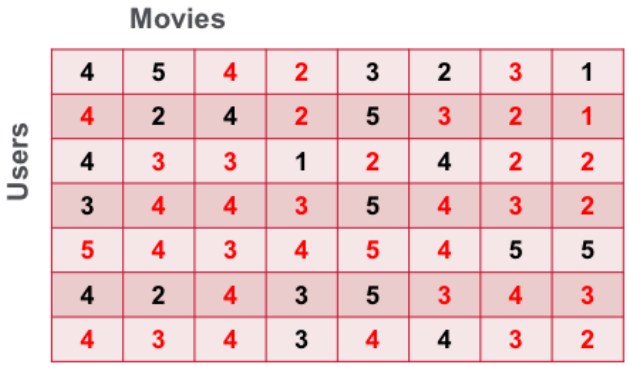
Ratings Matrix

Here's the overall goal:
Accurately predict every user's rating for the movies they haven't watched yet.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Overall goal: Complete the rating matrix
- The rating matrix is your central data structure.



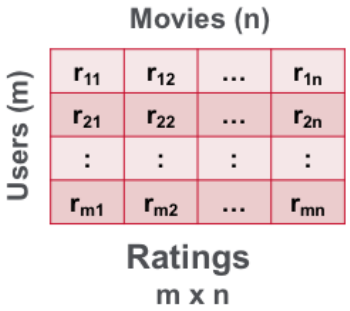
Ratings Matrix

If we do this right, we can suggest the most suitable movies to watch next to each user individually.
This of course can apply to lots of things – in our case it's movies.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- There are a total of “m” users (rows)
- And a total of “n” movies (columns)



We’ll shrink this down for the sake of keeping things simple.

What we have are some number of users => “m” rows
And some number of movies => “n” columns

The overall ratings matrix is m x n (rows x columns)

I used an explanation from Cambridge Coding Academy’s “Predicting User Preferences in Python using Alternating Least Squares” tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Break out users – We will denote them with the letter "P"

Let's break off the users – We know that there are m number of them, that's a given

But there's something we don't know about these users
There are certain influences that might cause these users to rate movies higher or lower
It could be **demographics** - age-related, gender, profession, whatever.
The point is, **we don't know what they are**, we just know they're present in the background somewhere.

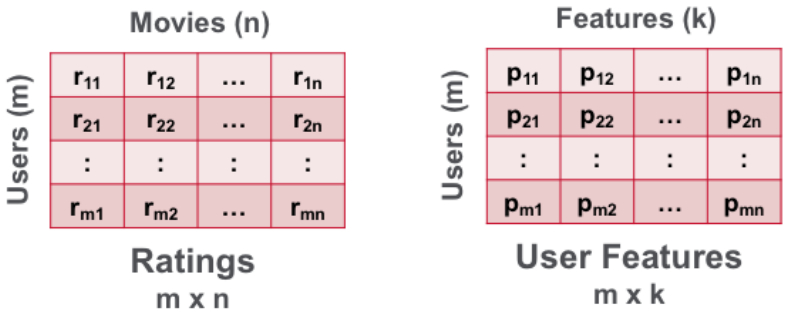
I used this reference regarding latent variables:
http://stats.stackexchange.com/questions/162585/predicting-score-in-the-presence-of-latent-variables

I used an explanation from Cambridge Coding Academy's

“Predicting User Preferences in Python using Alternating Least Squares” tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Break out users – We will denote them with the letter "P"
- The are influenced by "features" – a total quantity of "k"



Now we'll introduce a new variable called **"featues"**
Algorithms call these different things, such as "**latent factors**" or "**latent variables** or "**hidden features**"
For the sake of expediency, we'll call them "**Features**"
We also don't know how many there are, so we'll just throw them in the matrix for now
We'll assign an arbitrary number of to represent their quantity
In other words, we're going to assume that there are **K features** for each user that might influence their rating in some way.

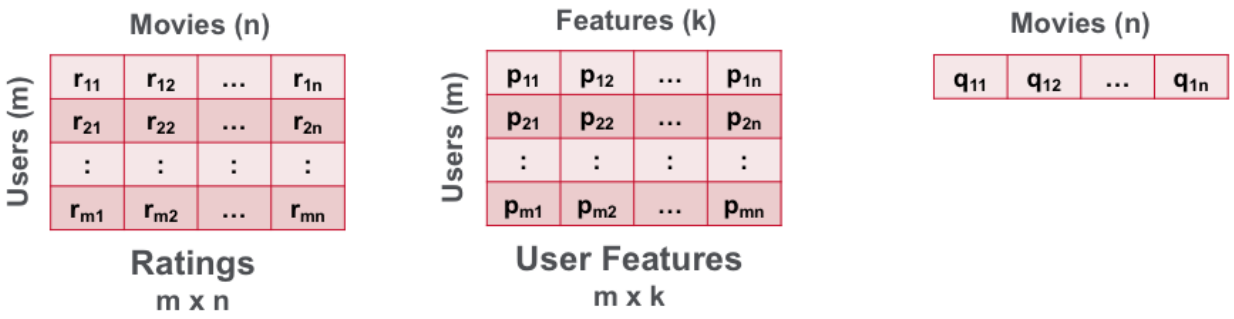
I used this reference regarding latent variables:
http://stats.stackexchange.com/questions/162585/predicting-score-in-the-presence-of-latent-variables

I used an explanation from Cambridge Coding Academy's

“Predicting User Preferences in Python using Alternating Least Squares” tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Break out movies – We will denote them with the letter "Q"



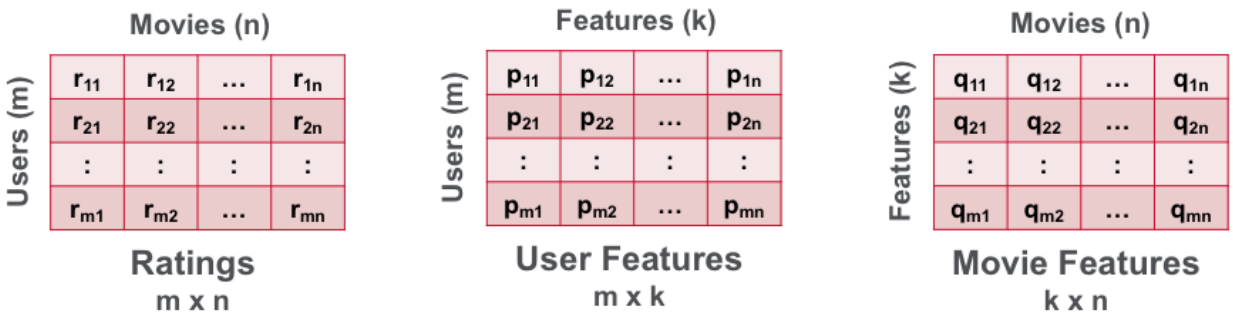
Likewise, let's break off the movies – We know that there are n number of them, that's a given

Like users, there's something we don't know about these movies
It's those same **features** that we just talked about…

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Break out movies – We will denote them with the letter "Q"
- The are also influenced by "features" – a total quantity of "k"



 29

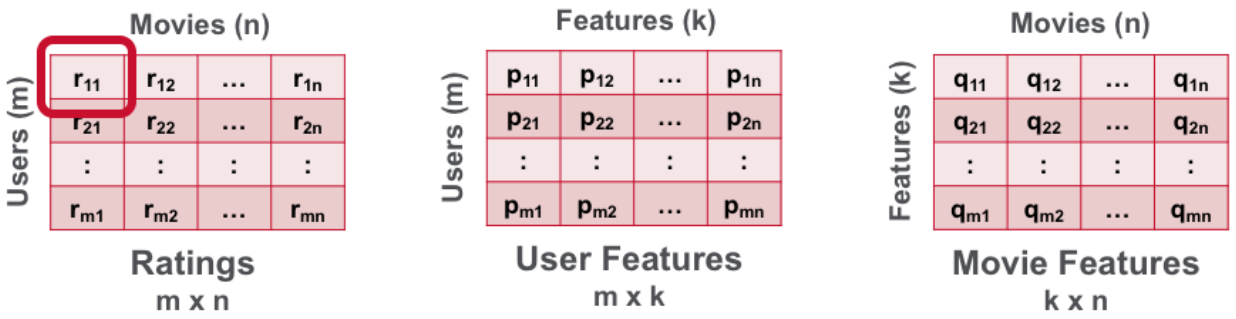
So we'll make the same assumption for the movies:
We'll assume that the same features that influence users to give certain ratings to certain movies would likewise be applicable
For example if we knew 25-29 year old men from California tend to rate certain types of movies a certain way, we could extrapolate the other direction
And likewise say certain types of movies are rated a certain way by 25-29 year old men from California

So these factor matrices represent hidden features which the algorithm tries to discover.
One matrix tries to describe the latent or hidden features of each user, and one tries to describe latent properties of each movie.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Ratings are the **sum of user features** weighted by **movie features**

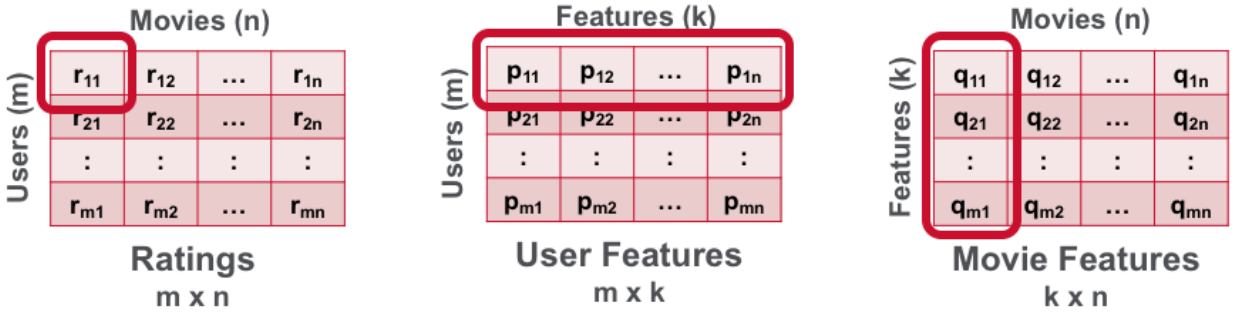


Each of the **ratings** is actually the the sum of latent user features, weighted by the respective latent movie features. You get that by computing the **dot product** of the user and movie features matrices

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Ratings are the **sum of user features** weighted by **movie features**
- Dot Product $r_{11} = (p_{11} \text{ X } q_{11}) + (p_{12} \text{ X } q_{21})$ and so on...



 31

Here's how **dot products** work:
A dot product is composed of the corresponding **user features** row and the first **movie features** column

It looks like this: R11 = (P11 X Q11) + (P2 X Q21) and so on.

But there's a problem…

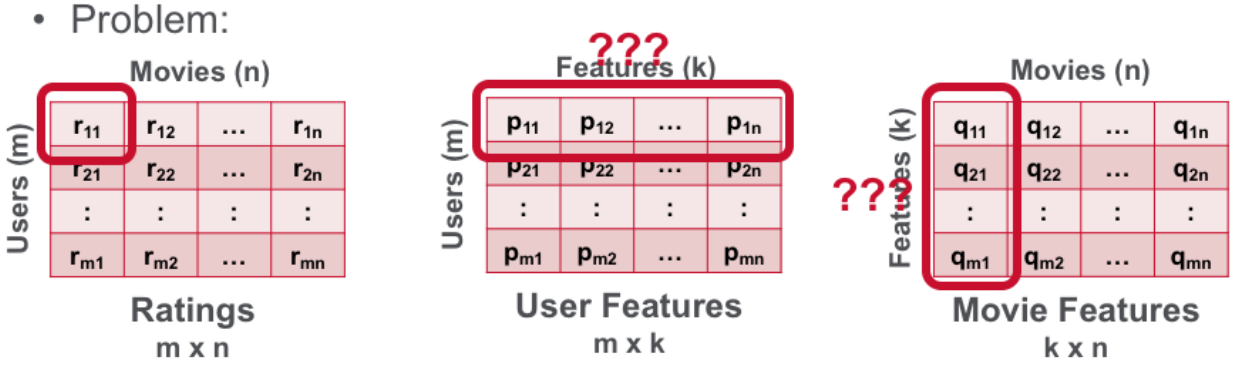
The concept of **dot product** is explained here:
https://www.mathsisfun.com/algebra/matrix-multiplying.html

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:

http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Rating are the **sum of user features** weighted by **movie features**
- Dot Product $r_{11} = (p_{11} \text{ X } q_{11}) + (p_{12} \text{ X } q_{21})$ and so on...
- Problem:



 32

Problem:
We don't know what the features are, or how many we have

For that, we'll use the trick of **alternating least squares**.

The concept of **dot product** is explained here:
https://www.mathsisfun.com/algebra/matrix-multiplying.html

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- We already know *some* of the movie ratings

 33

Here's the concept:
We already know some of the movie ratings – remember these?

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here:
http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

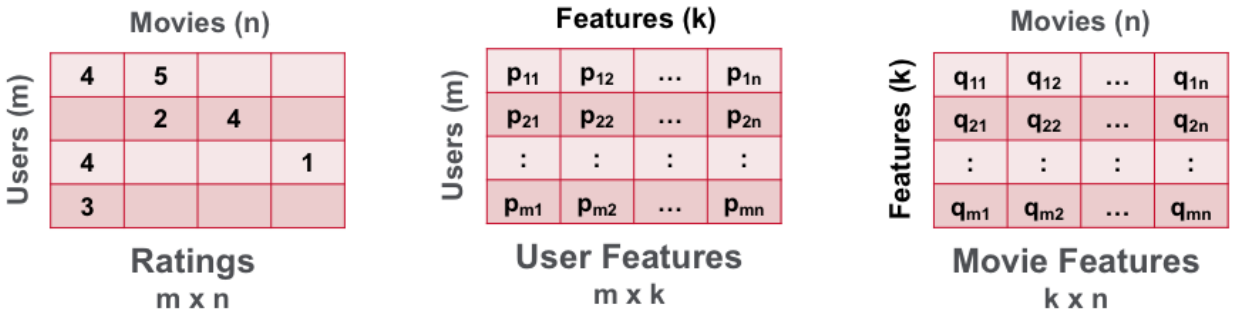
- We already know *some* of the movie ratings
- Our goal is to fill in the ones that we don't know

Our goal here is to fill in the ones that we don't know.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

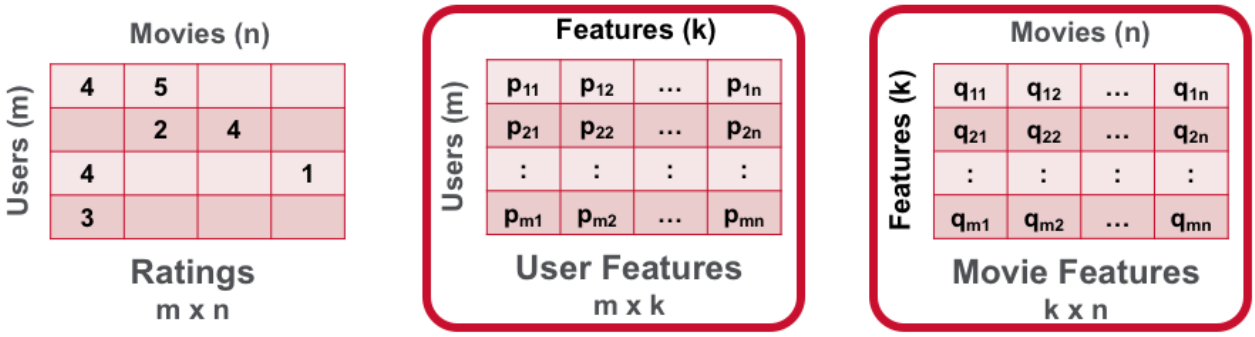
- We'll use what we do know to solve for what we don't know



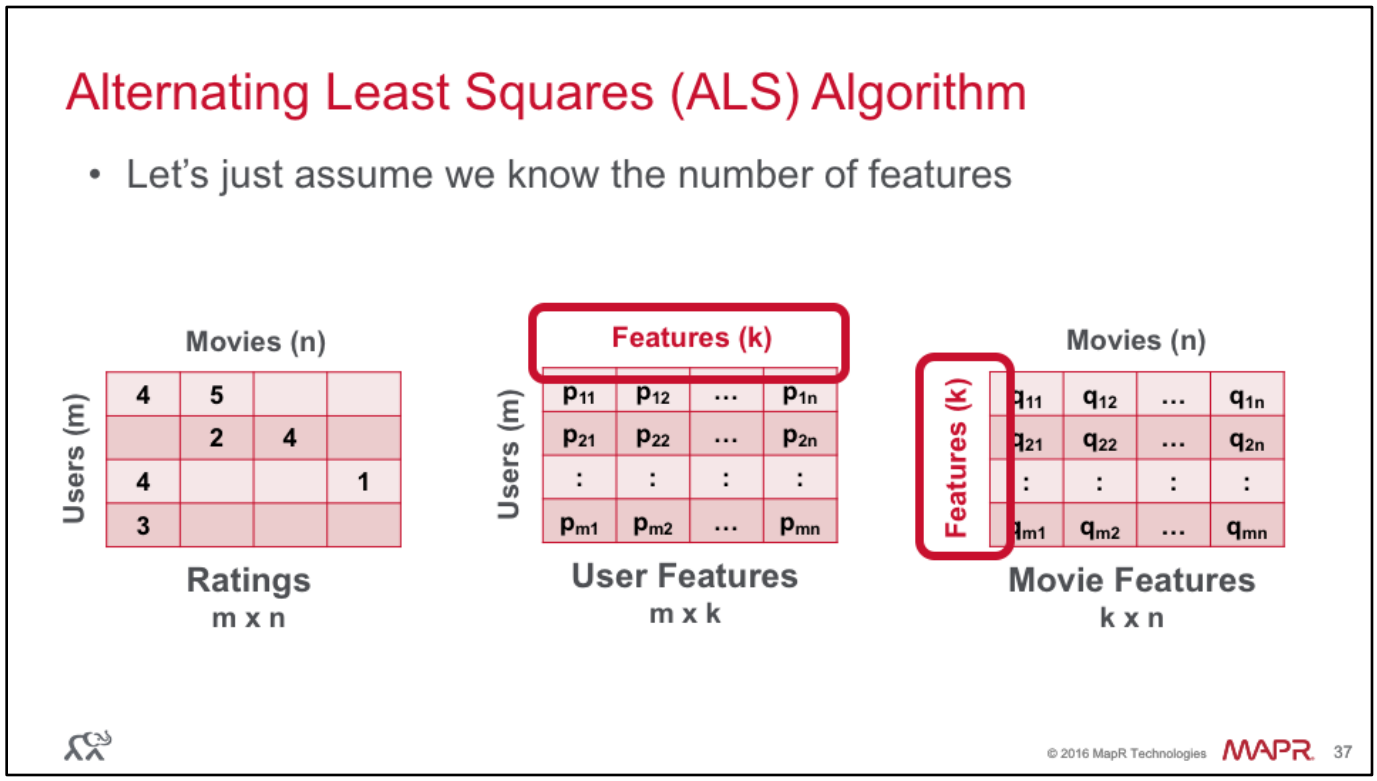
I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- We'll use what we do know to solve for what we don't know
- Namely, the **User Features** and **Movie Features** matrices



I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

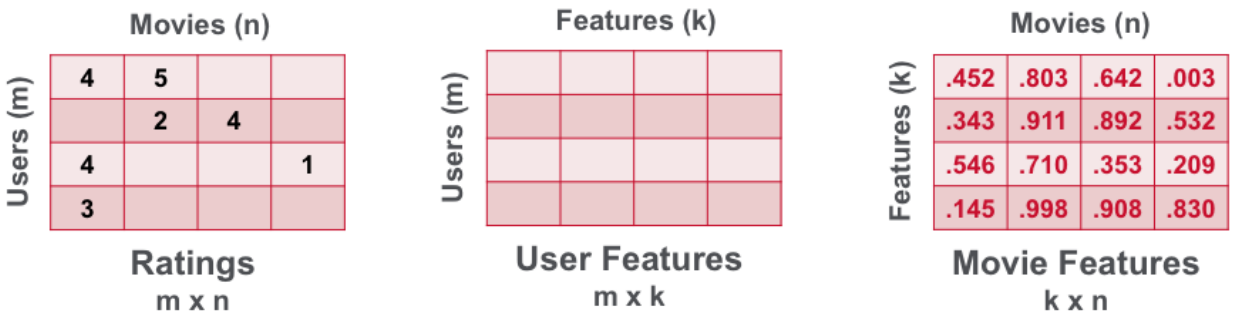


We'll start by assuming we know we know the number of features

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

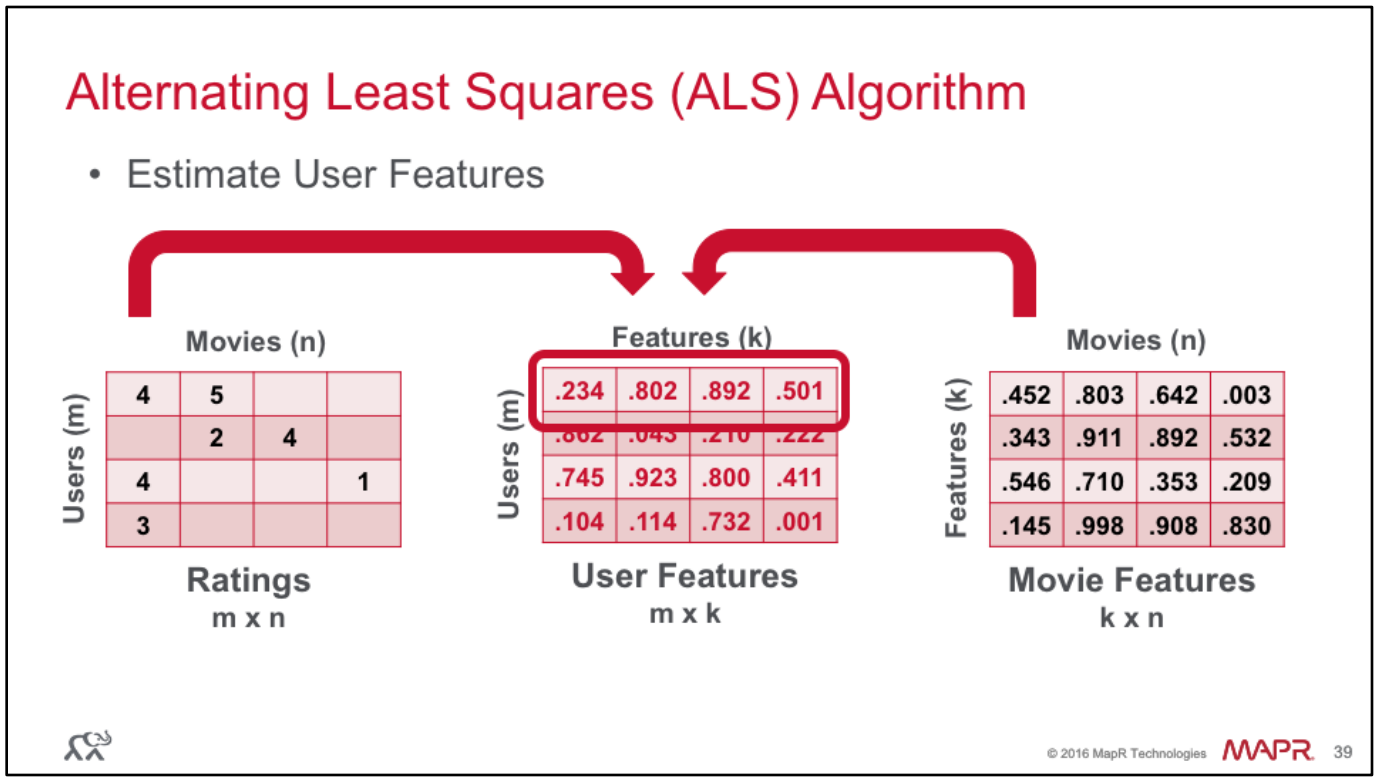
Alternating Least Squares (ALS) Algorithm

- Let's just assume we know the number of features
- We'll "fix" the Movie Features matrix Q with random values:



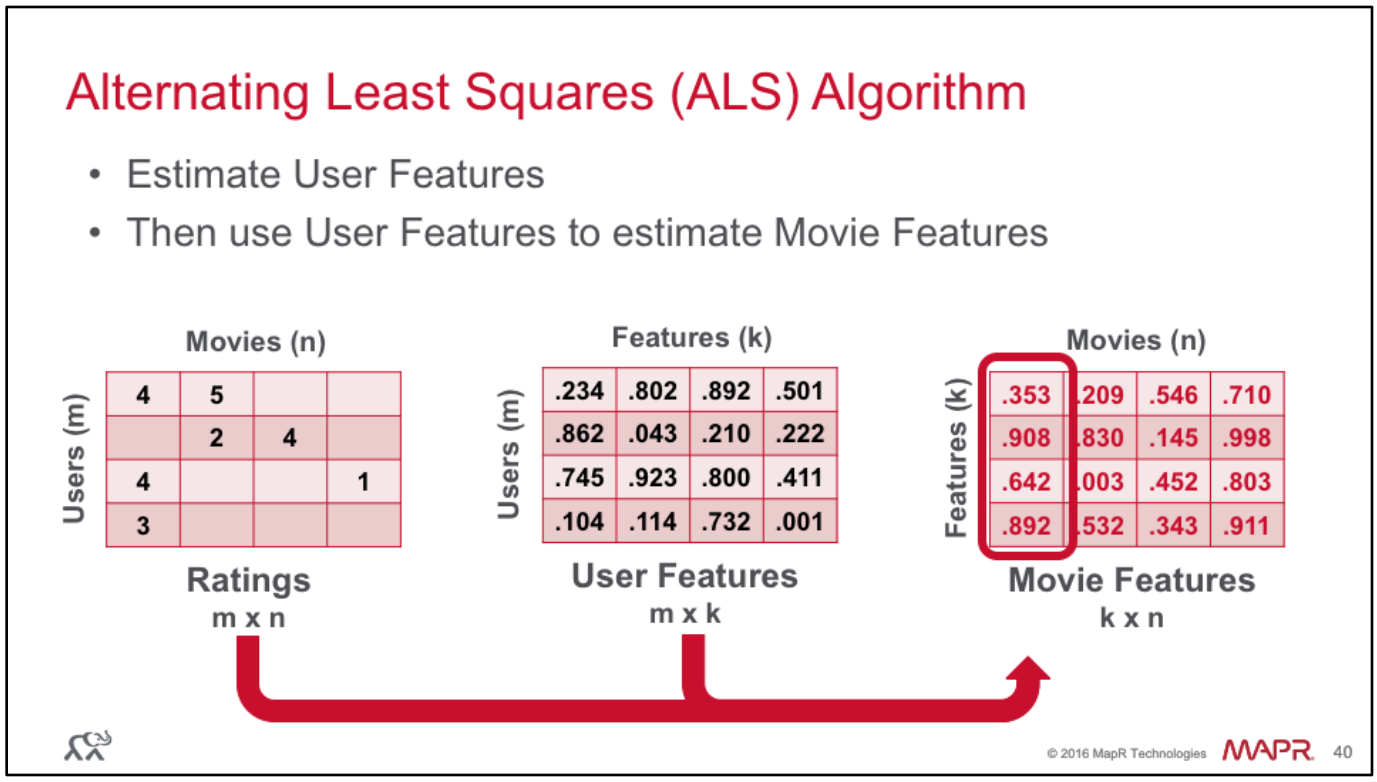
And we'll fill the **Movie Features** matrix with random values

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares



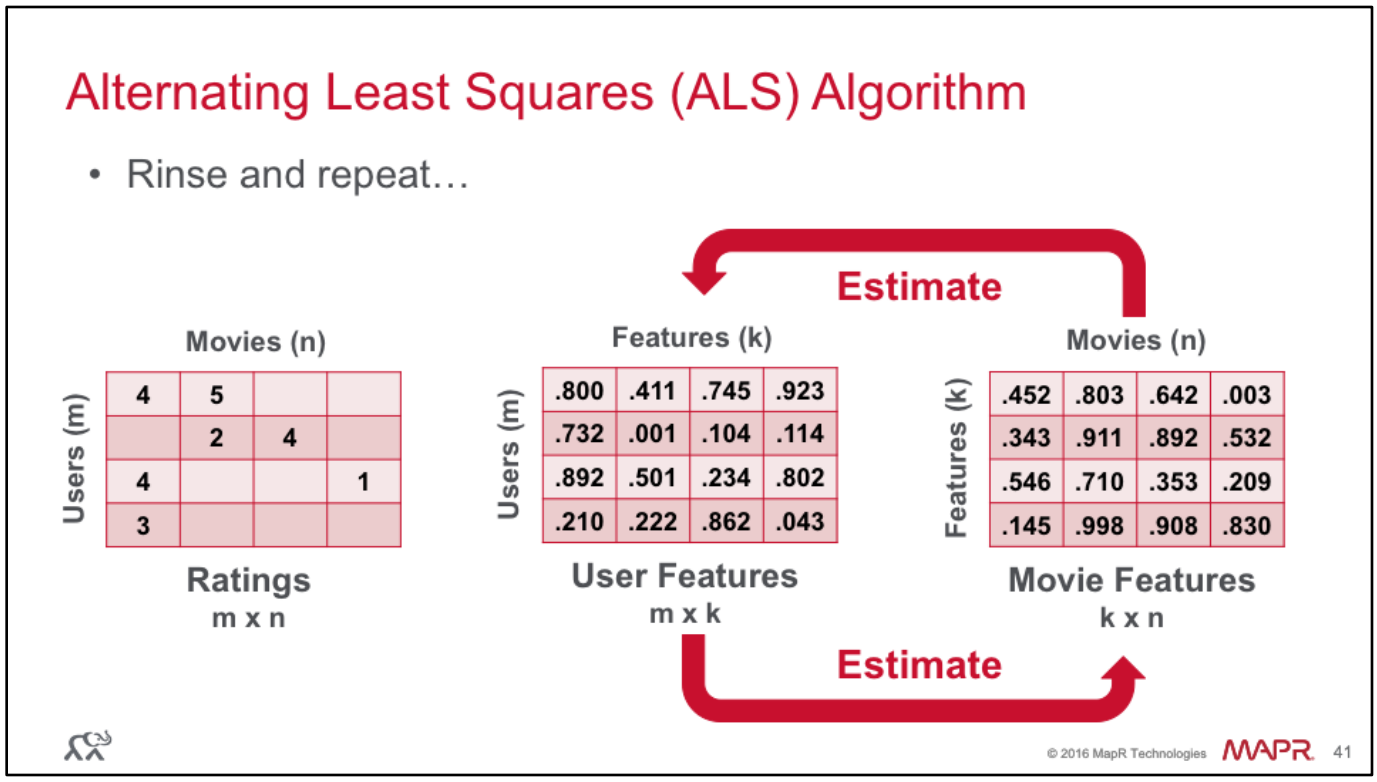
We'll then estimate the **User Features** matrix for every user, row by row, using the non-empty cells from the **Ratings Matrix** and the random values from the **Movie Features** matrix.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares



We'll then estimate the **Movie Features** matrix for every movie, column by column, using the non-empty cells from the **Ratings Matrix** and the values from the **User Features** matrix.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares



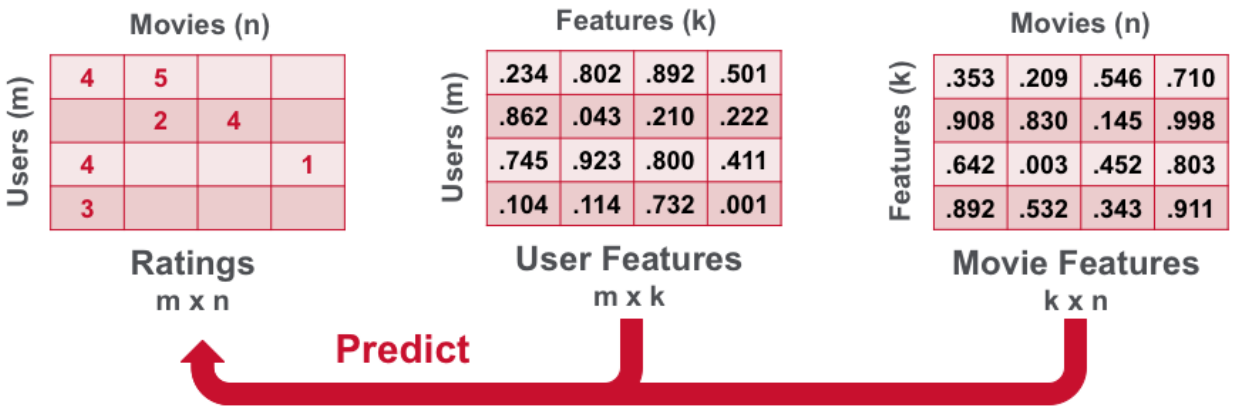
We'll repeat this procedure some number of times

This alternation between which matrix to optimize is where the "**alternating**" in the name comes from.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

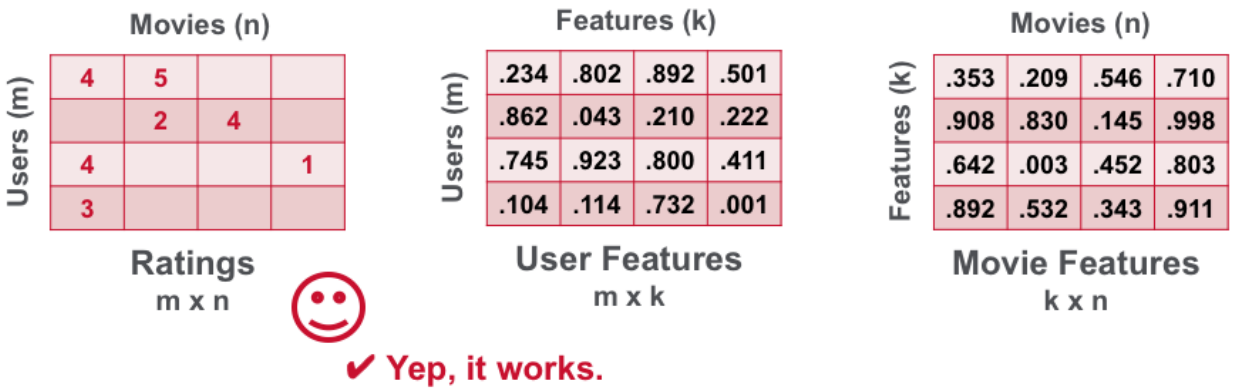
- Eventually we'll have a model for predicting the ratings we know



I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Eventually we'll have a model for predicting the ratings we know
- We call this **Convergence**

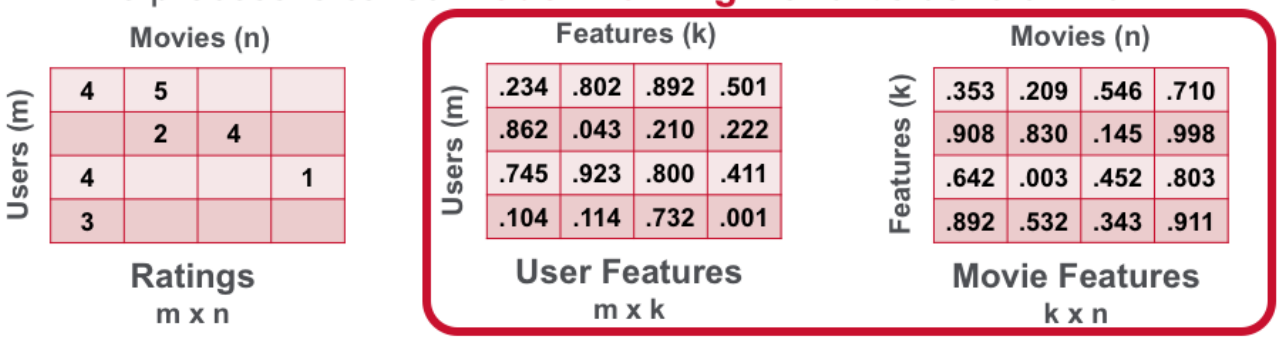


✔ Yep, it works.

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

Alternating Least Squares (ALS) Algorithm

- Eventually we'll have a model for predicting the ratings we know
- We call this **Convergence**
- The process is called **Model Training** – and it's done *offline*

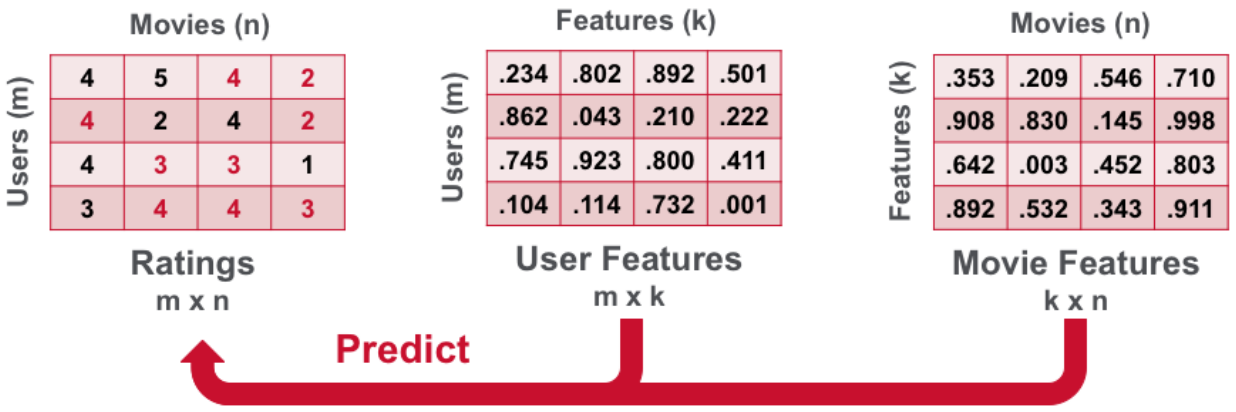


 MAPR

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares

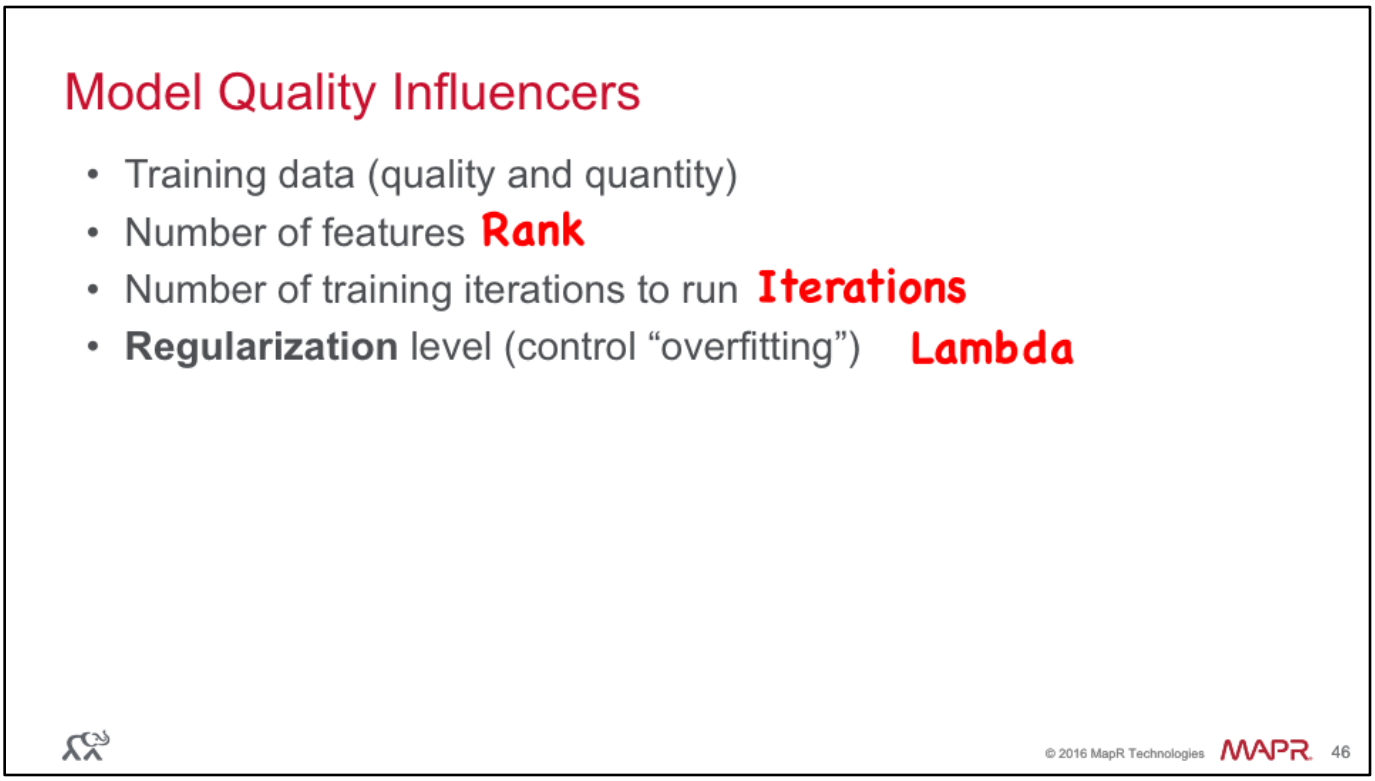
Alternating Least Squares (ALS) Algorithm

- Now we can use the model to **predict what we don't know**:



 MAPR 45

I used an explanation from Cambridge Coding Academy's "Predicting User Preferences in Python using Alternating Least Squares" tutorial located here: http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares



So we have things that will influence how good our model is

Regularization is a technique used in an attempt to solve the overfitting problem in statistical models.

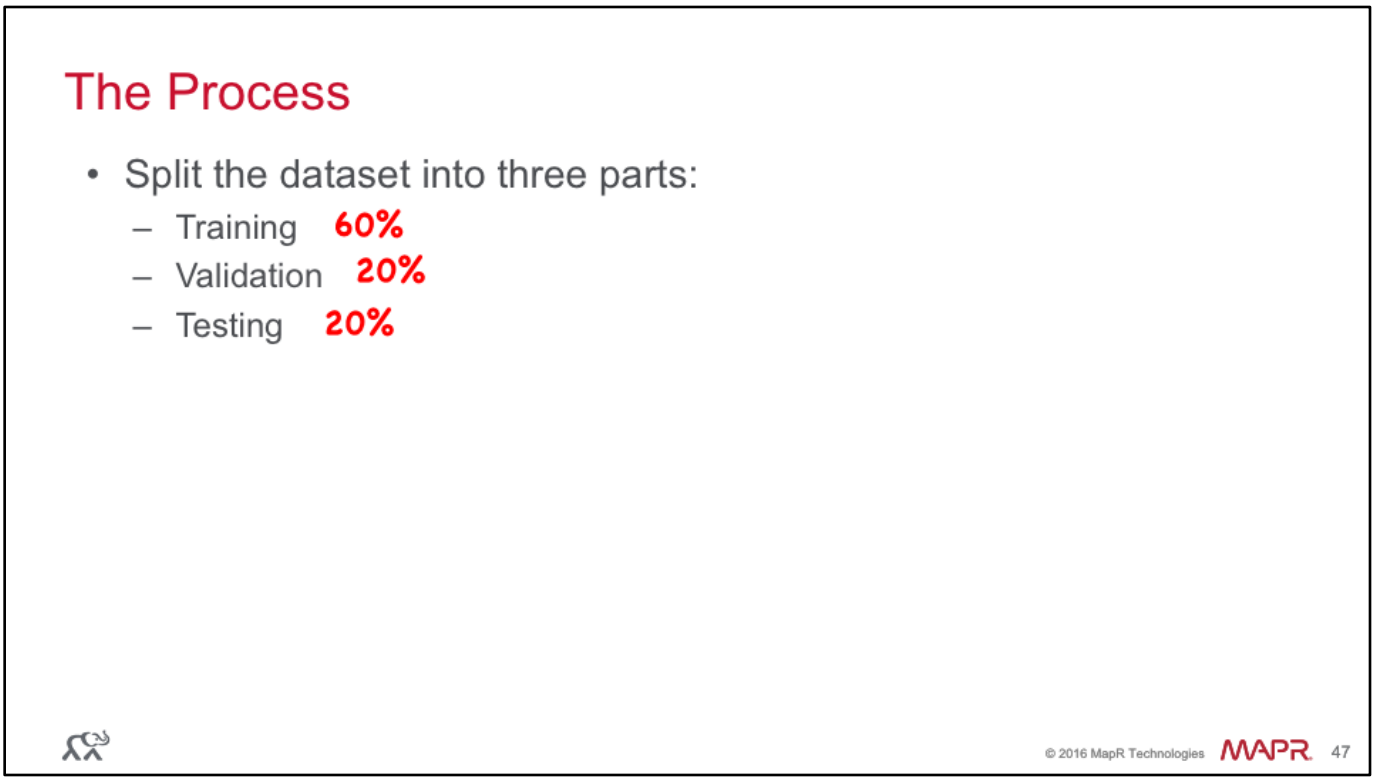
Here's how it works, in a nutshell:
Let's assume we only have two features that influence movie ratings – age and gender. That's it.
Well probably this model will fail because it's too simple.
So let's throw in more demographics such as profession, what part of the country they're from, education level, etc.
This makes the model more **interesting** and **complex**.
Our model will do better in some ways, but worse in others due to this concept of overfitting
This is because it's sticking too much to the data – in other words, it can't generalize

Which means we're being influenced by "background noise"
So overfitting happens when the model works well on training data using known noise but doesn't perform well on the actual testing data.
This isn't acceptable.

So we apply a regularization factor to control the model's complexity, and thus help prevent overfitting
The higher the regularization factor, the lower the overfitting, but the greater the bias.
If you're a visual person, imagine smoothing a regression line so that it's less influenced by noisy data points.

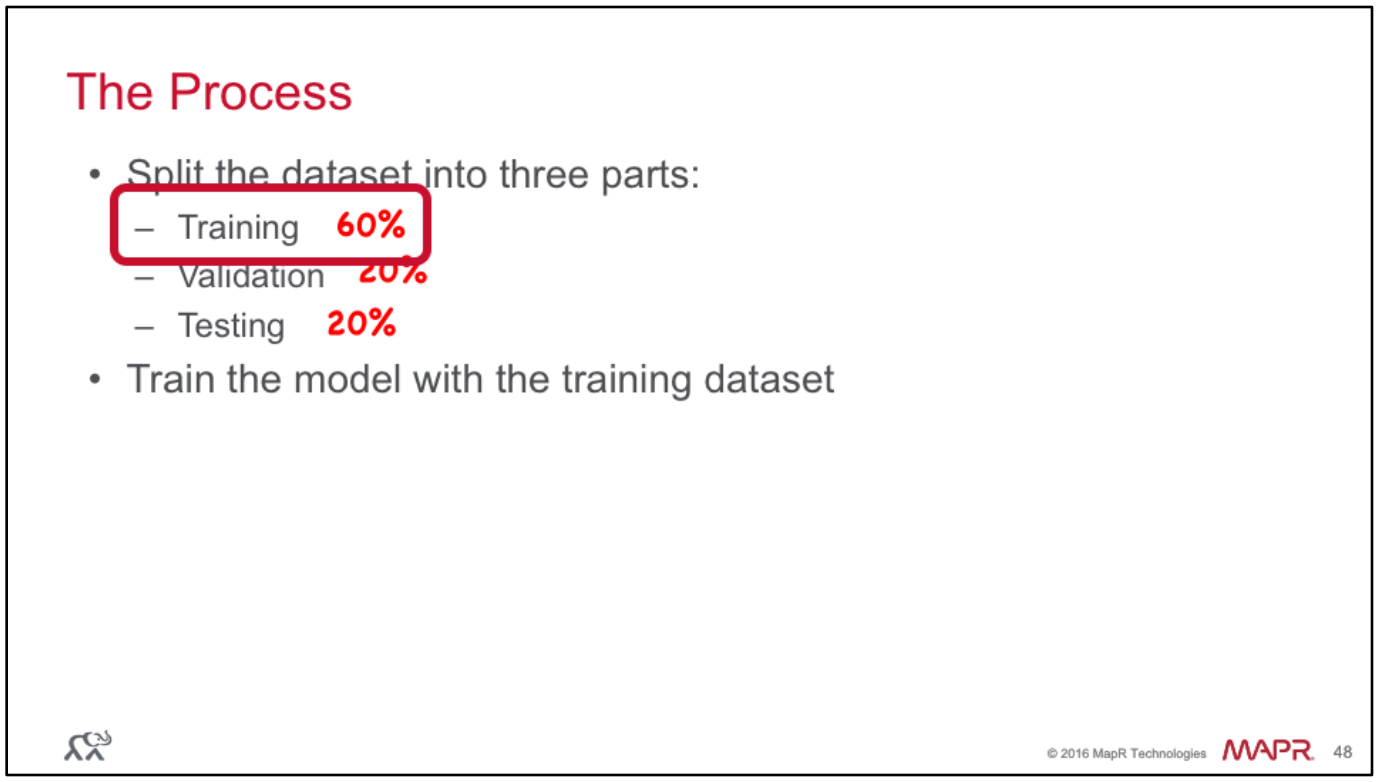
Incidentally, three of these are actually parameters into the algorithm
Rank - Starting at 5 and increasing by 5 until the recommendation improvement rate slows down, memory and CPU permitting, is a good approach
Iterations – Likewise, 5, 10, and 20 iterations for various combinations of rank and lambda are good starting points
Lambda - Values of 0.01, 1 and 10 are good values to test.

I used an explanation from Quora here:
https://www.quora.com/What-is-regularization-in-machine-learning
And here:
https://cloud.google.com/solutions/recommendations-using-machine-learning-on-compute-engine

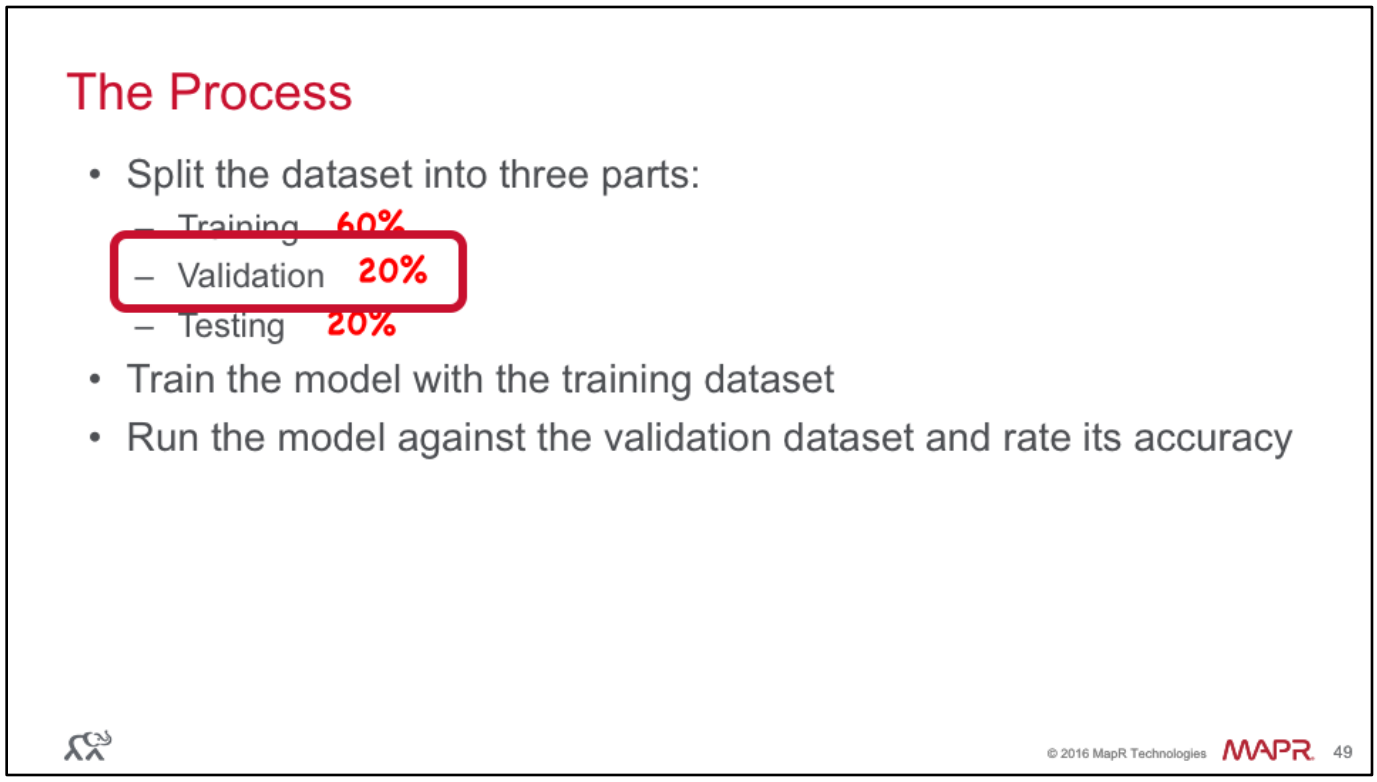


Split your dataset into three pieces:

- One for training
- One for validation
- One for testing



Train the model with the training data – This is where the model gets created



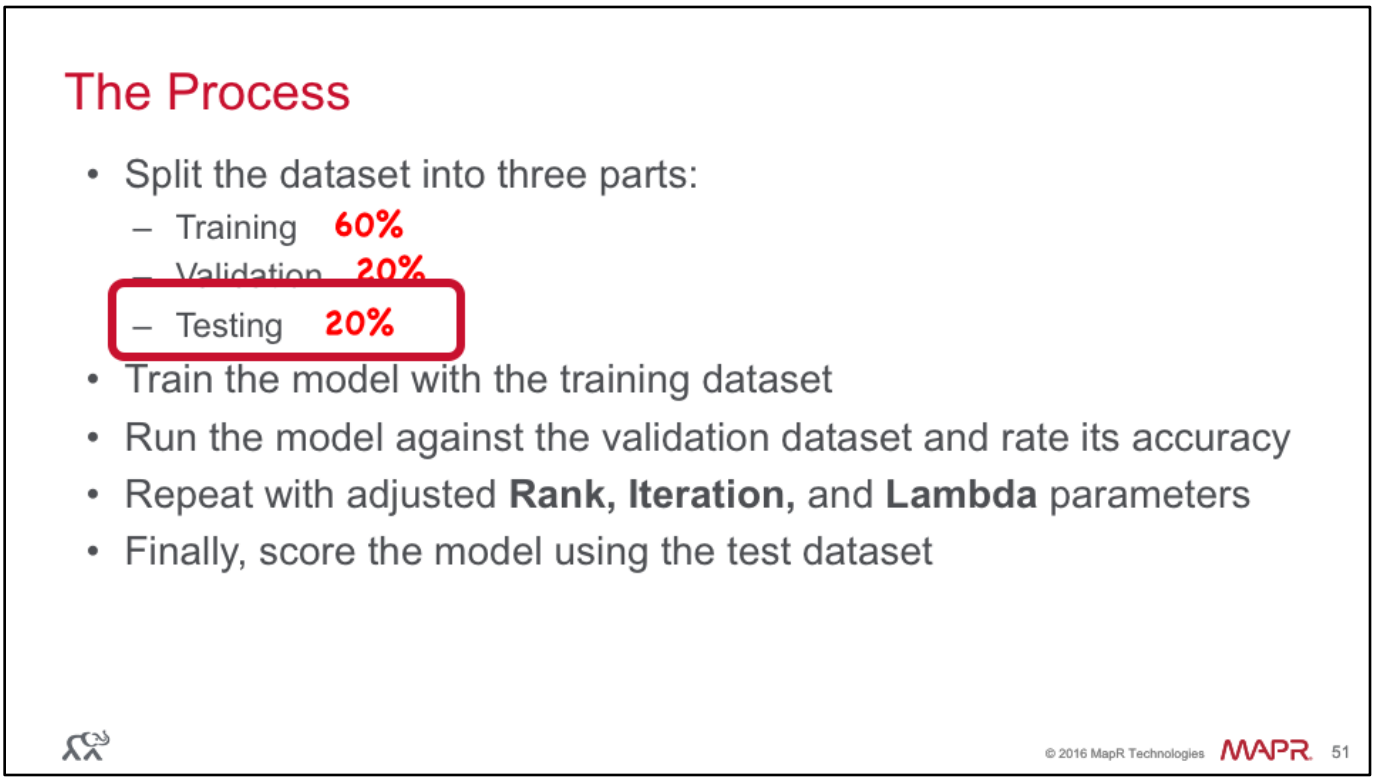
Run the model against the validation data

(Why not run it against the training data? Because we already know it works pretty well with that data set)

The Process

- Split the dataset into three parts:
  - Training 60%
  - Validation 20%
  - Testing 20%
- Train the model with the training dataset
- Run the model against the validation dataset and rate its accuracy
- Repeat with adjusted **Rank, Iteration,** and **Lambda** parameters

Change your **rank, iteration,** and **lambda** values and re-run
Did your accuracy improve?
Do that over and over again until you're happy with the result



Finally, run it against the test dataset and check its accuracy. That's your final score.
If score isn't good enough then you might add additional training data or refine parameter adjustments.

The Process

- Split the dataset into three parts:
  - Training 60%
  - Validation 20%
  - Testing 20%
- Train the model with the training dataset
- Run the model against the validation dataset and rate its accuracy
- Repeat with adjusted **Rank, Iteration,** and **Lambda** parameters
- Finally, score the model using the test dataset
- What's missing?

There's one more thing we need here – a standard of measurement for rating the model's accuracy

Root Mean Square Error (RMSE)

- A frequently-used performance measurement
- Evaluates difference between **predicted** and **actual** values

For our example we'll use something called **Root Mean Square Error**
This is a frequently used measure of the differences between the values predicted by a model and the values actually observed.

I used an explanation from Wikipedia here:
https://en.wikipedia.org/wiki/Root-mean-square_deviation

Root Mean Square Error (RMSE)

- A frequently-used performance measurement
- Evaluates difference between **predicted** and **actual** values
- In mathematical terms:

> *"RMSE is the **sample standard deviation** of the **differences** between **predicted** and **observed** values"*

In mathematical terms, it represents the **sample standard deviation** of the **differences** between **predicted** and **observed** values.

I used an explanation from Wikipedia here:
https://en.wikipedia.org/wiki/Root-mean-square_deviation

Root Mean Square Error (RMSE)

- A frequently-used performance measurement
- Evaluates difference between **predicted** and **actual** values
- In mathematical terms:

> *"RMSE is the* ***sample standard deviation*** *of the* ***differences*** *between* ***predicted*** *and* ***observed*** *values"*

- We will use it as a **cost function** to evaluate our model
- Our goal will be to keep the cost as low as possible.

I used an explanation from Wikipedia here:
https://en.wikipedia.org/wiki/Root-mean-square_deviation

Demo #2: Under The Hood With MLlib ALS

Import library for MLlib recommendation data types

```
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
```

Import library for MLlib recommendation data types

```
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
```

Create Ratings RDD, Load Data Frame, Register

```
def parseRating(str: String): Rating = {
      val fields = str.split("::")
      Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
 }

val ratingsRDD = sc.textFile("ratings.dat").map(parseRating).cache()

val ratingsDF = ratingsRDD.toDF

ratingsDF.registerTempTable("ratings")
```

 MAPR 58

Note the use of the **cache** method here – I want to keep this in memory

Define the RMSE function

```
import org.apache.spark.rdd._
def computeRmse(model: MatrixFactorizationModel, data: RDD[Rating], n: Long): Double = {
    val predictions: RDD[Rating] = model.predict(data.map(x => (x.user, x.product)))
    val predictionsAndRatings = predictions.map(x => ((x.user, x.product), x.rating))
    .join(data.map(x => ((x.user, x.product), x.rating))).values
    math.sqrt(predictionsAndRatings.map(x => (x._1 - x._2) * (x._1 - x._2)).reduce(_ + _) / n)
}
```

 59

I borrowed this one from an AWS blog here: https://blogs.aws.amazon.com/bigdata/post/Tx6J5RM20WPG5V/Building-a-Recommendation-Engine-with-Spark-ML-on-Amazon-EMR-using-Zeppelin

Split Ratings RDD into Training/Validation/Test RDDs

```
val splitsRDD = ratingsRDD.randomSplit(Array(0.6, 0.2, 0.2), 0L)
val trainingRDD = splitsRDD(0).cache()
val validationRDD = splitsRDD(1).cache()
val testRDD = splitsRDD(2).cache()
val numTraining = trainingRDD.count()
val numValidation = validationRDD.count()
val numTest = testRDD.count()
```

60%, 20%, 20%

 60

Note that this is a 60% / 20% / 20% split

Validation Runs

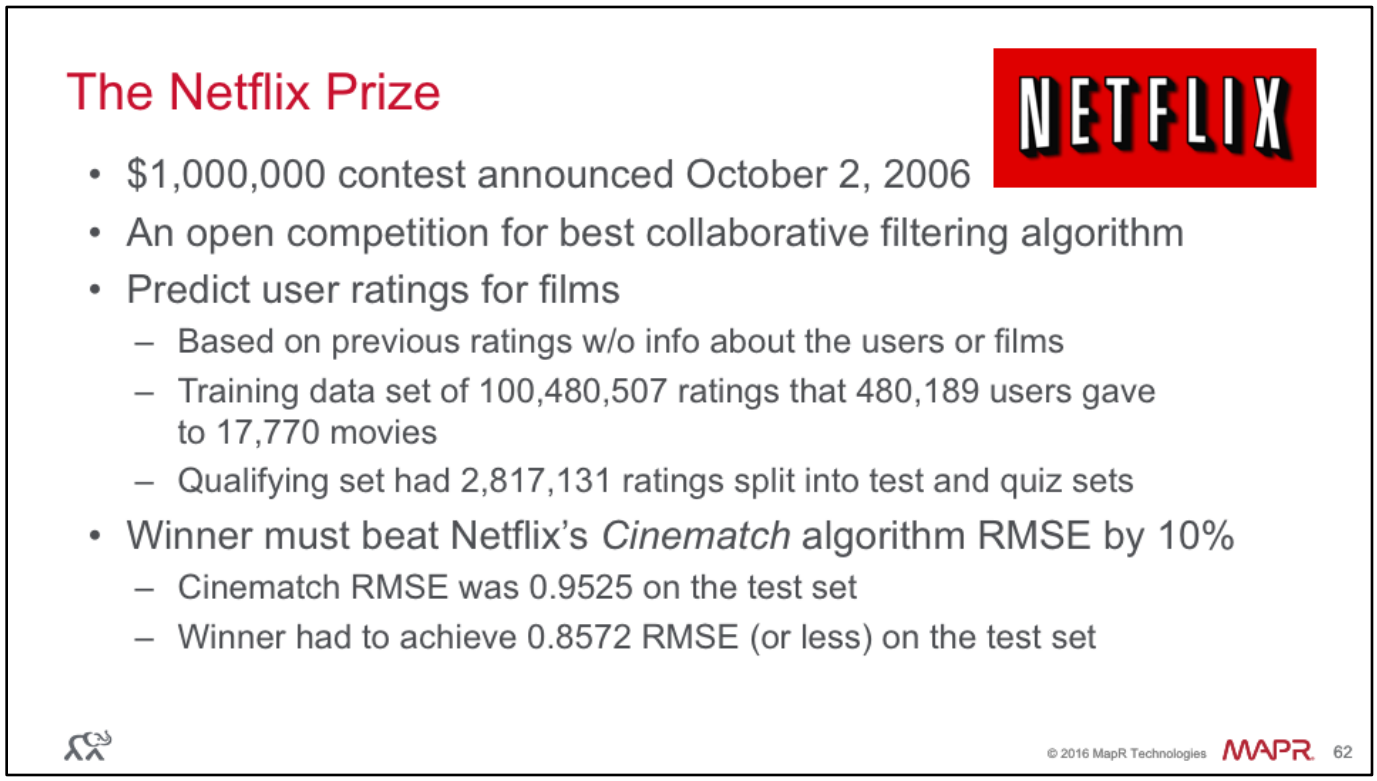
```
val ranks = List(5, 10 15)
val iters = List(10, 20)
val lambdas = List(0.1, 1.0)
var bestModel: Option[MatrixFactorizationModel] = None
var bestRmse = Double.MaxValue
var bestRank = 0
var bestIter = -1
var bestLambda = -1.0
for (rank <- ranks; iter <- iters; lambda <- lambdas) {
   val model = ALS.train(trainingRDD, rank, iter, lambda)
   val validationRmse = computeRmse(model, validationRDD, numValidation)
   println(s"RMSE = $validationRmse for the model trained with rank = $rank, lambda = $lambda, and
iter = $iter.")
   if (validationRmse < bestRmse) {
      bestModel = Some(model)
      bestRmse = validationRmse
      bestRank = rank
      bestLambda = lambda
      bestIter = iter
   }
}
```

12 runs (3 x 2 x 2)

 MAPR

I've got three parameters, with a variety of values for each, for a total of 12 runs
The model will train
At the end of each run it'll tell us the RMSE

The lowest RMSE will be our final test run parameters.



Thought it might be fun to take a look at another movie ratings project called "The Netflix Prize"

There were fairly sophisticated rules around how yearly progress prizes would be awarded

To win a progress or grand prize a participant had to provide source code and a description of the algorithm to the jury. Following verification the winner also had to provide a non-exclusive license to Netflix.
Netflix would publish only the description, not the source code, of the system.
A team could choose to not claim a prize, in order to keep their algorithm and source code secret.

Once one of the teams succeeded to improve the RMSE by 10%

or more, the jury would issue a last call, giving all teams 30 days to send their submissions.
Only then, the team with best submission was asked for the algorithm description, source code, and non-exclusive license, and, after successful verification; declared a grand prize winner.

The contest would last until the grand prize winner was declared.
Had no one received the grand prize, it would have lasted for at least five years (until October 2, 2011).
After that date, the contest could have been terminated at any time at Netflix's sole discretion.

Two progress prizes were awarded in 2007 and 2008
Finally, on September 18, 2009, Netflix announced team "BellKor's Pragmatic Chaos" as the prize winner

References:
https://en.wikipedia.org/wiki/Netflix_Prize
https://www.techdirt.com/blog/innovation/articles/20120409/03412518422/why-netflix-never-implemented-algorithm-that-won-netflix-1-million-challenge.shtml

The Netflix Prize

Here's a picture of the BellKor's Pragmatic Chaos team

Their algorithm had achieved a Test RMSE of 0.8567
The team consisted of two Austrian researchers from a consulting firm,
Two researchers from AT&T Labs
A Yahoo! reseacher
And two researchers from Pragmatic Theory

On December 17, 2009, four Netflix users filed a class action lawsuit against Netflix, alleging that Netflix had violated U.S. fair trade laws and the Video Privacy Protection Act by releasing the datasets.
There was public debate about privacy for research participants.
On March 19, 2010, Netflix reached a settlement with the plaintiffs, after which they voluntarily dismissed the lawsuit.

Postnote:
Netflix never implemented the winning solution!
They did, however, make use of a blend of two algorithms that members of the grand prize winning BellKor team had submitted to win the first $50K progress prize in 2007 (they were the AT&T Labs and Yahoo! Researchers)
This submission had shown an 8.43% improvement.

The grand prize winning algorithm? Well, it wasn't worth the work it would take to get the additional marginal improvement.
Plus, by then Netflix's business had shifted to online streaming
And recommendations for streaming videos is different than for rental viewing a few days later.

In other words, the shift from delayed gratification to instant gratification makes a difference in the kinds of recommendations that work

References:
https://en.wikipedia.org/wiki/Netflix_Prize
https://www.techdirt.com/blog/innovation/articles/20120409/03412518422/why-netflix-never-implemented-algorithm-that-won-netflix-1-million-challenge.shtml

 64

Speaking of movies, here's one you might consider watching at some point
This one came out in 2014
It's a loose biography of **Alan Turing**
The reason I bring it up is because **Turing** pioneered the concept of **solving mathematical algorithms** using general-purpose computers
He's widely considered the father of artificial intelligence, from which machine learning has evolved

The movie takes a very liberal theatrical license with the story it tells of Turing's life, but it does demonstrate one of the first real-world use cases
(namely, saving people's lives)

Next Steps

- Save the model
- Run against additional test data until RMSE is acceptable
  - Training and parameter optimization can be automated
  - Can also automate periodic data enrichment
- Engine Deployment options:
  - REST Service
  - Package as a Java, Scala, or Python application
  - Implement as a real-time streaming application

 65

Here are some potential next steps

First, you'll want to save the model
Spark makes this pretty straightforward

Next, you may do some additional test runs
A lot of this can be automated

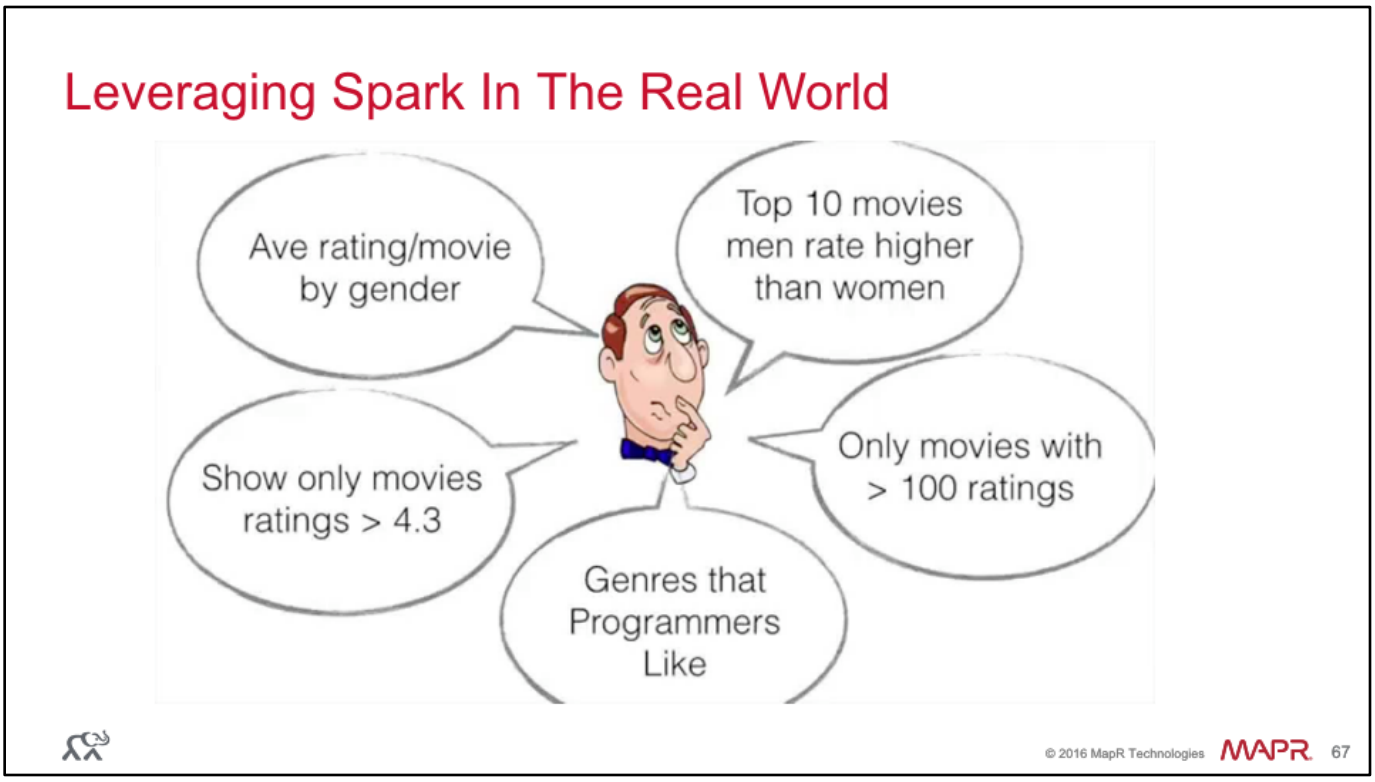
You've also got several deployment options at your disposal

Really, it depends on your use case for the prediction model

This will be a quick demonstration of how to leverage Spark SQL ODBC/JDBC connectivity to query recommendation data using familiar tools

I've staged a demo recommendations table in a Parquet file

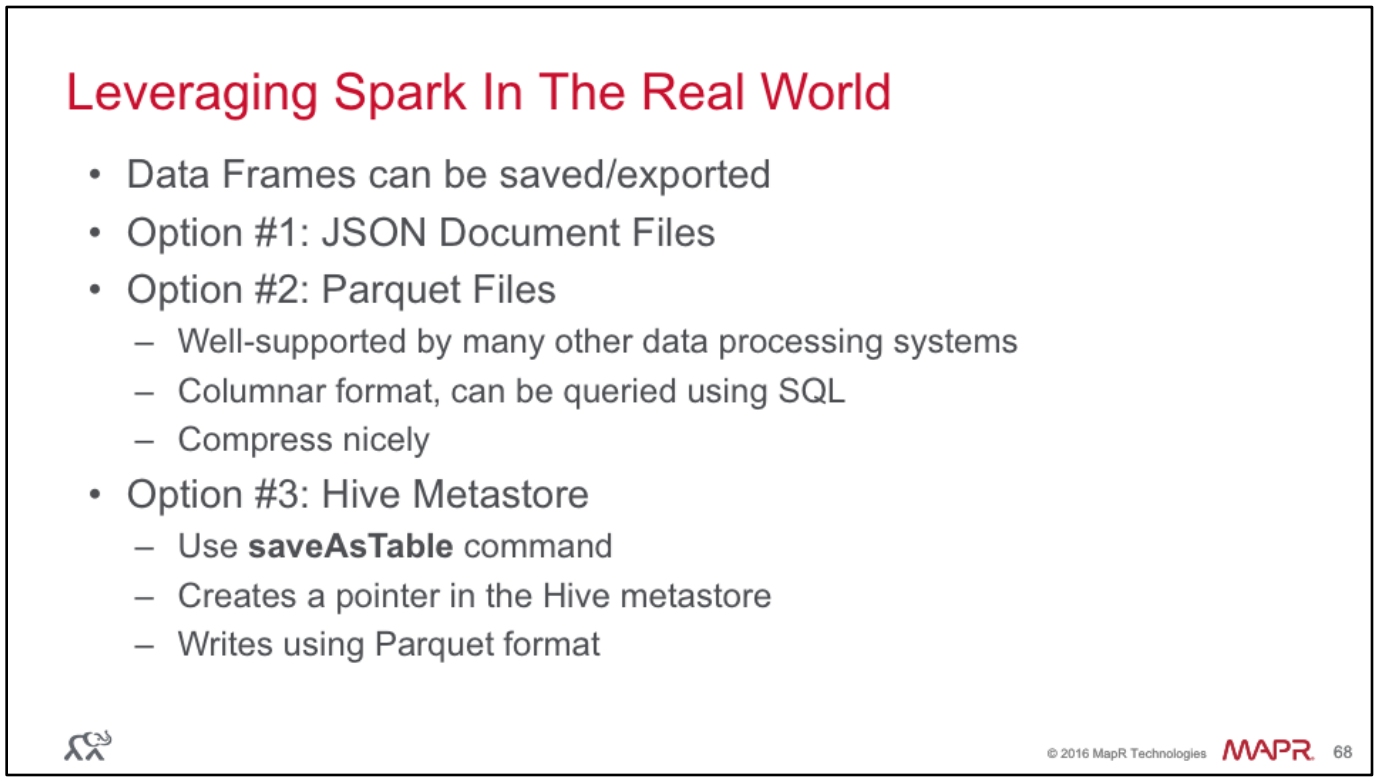
This table contains predicted ratings of all unwatched movies for a relatively small subset of the users (200 to be exact).



Spark is a great tool for data scientist and engineers
We could play with this dataset all day long

But the truth is, **Spark isn't an end-user tool**
We need to expose the datasets to the outside world
There are some options here

I used this image from this video:
https://www.youtube.com/watch?v=BN4-GrZ1mGU



First, we'll assume your data is present within the context of a Data Frame
Remember this is structured data, just like a table in a relational database

We can save Data Frame as simple JSON document files in the filesystem.
This makes them reasonably portable.

A better option is to materialize dataframes as **Parquet files**
Parquet is a **columnar format** that is supported by many other data processing systems.
The nice thing about a columnar-formatted datastore is that they are well-suited for OLAP-type workloads against very large datasets (e.g., data warehouses)
This is because the data is stored as **columns** rather than rows

Another side benefit is that they tend to **compress very nicely as well**
You can still run SQL against these files

Still, a third option is to save them to a Hive metastore using the **saveAsTable** command
An existing Hive deployment is not necessary to use this feature.
Spark will create a default local Hive metastore using **Apache Derby** for you.
The **saveAsTable** command materializes the contents of a Data Frame and **creates a pointer to the data** in the Hive metastore
If you liked the sound of the columnar **Parquet** file format then you'll be happy to know that **saveAsTable** writes using Parquet as well
I'll demonstrate this third option with Tableau as my BI tool

I used this image from this video:
https://www.youtube.com/watch?v=BN4-GrZ1mGU
I referred to this Wikipedia entry:
https://en.wikipedia.org/wiki/Column-oriented_DBMS and this Spark documentation: https://spark.apache.org/docs/latest/sql-programming-guide.html

Saving Data Frames as Hive Metastore Tables

```
val options = Map("path" -> "/Users/Shared/Hive/movies")
moviesDF.write.options(options).mode("overwrite").saveAsTable("movies")

val options = Map("path" -> "/Users/Shared/Hive/users")
usersDF.write.options(options).mode("overwrite").saveAsTable("users")

val options = Map("path" -> "/Users/Shared/Hive/ratings")
ratingsDF.write.options(options).mode("overwrite").saveAsTable("ratings")
```

 69

As a side note, we looked at **registerTempTable** during the first demo
Remember that this creates an in-memory table that is scoped to the cluster in which it was created.
The data is stored using Hive's highly-optimized, in-memory columnar format.

The difference here is that **saveAsTable** creates a permanent, physical table in the filesystem

Notice that I'm writing to a shared area on my local laptop
I don't have a local Hive metastore, so as I mentioned earlier, Spark will create one for me using **Apache Derby**.

I referred to this document:
https://forums.databricks.com/questions/400/what-is-the-difference-between-registertemptable-a.html
And this Spark documentation:
https://spark.apache.org/docs/latest/sql-programming-guide.html

Saving Data Frames as Hive Metastore Tables

```
val options = Map("path" -> "/Users/Shared/Hive/movies")
moviesDF.write.options(options).mode("overwrite").saveAsTable("movies")

val options = Map("path" -> "/Users/Shared/Hive/users")
usersDF.write.options(options).mode("overwrite").saveAsTable("users")

val options = Map("path" -> "/Users/Shared/Hive/ratings")
ratingsDF.write.options(options).mode("overwrite").saveAsTable("ratings")
```

Start the Thrift JDBC/ODBC Server

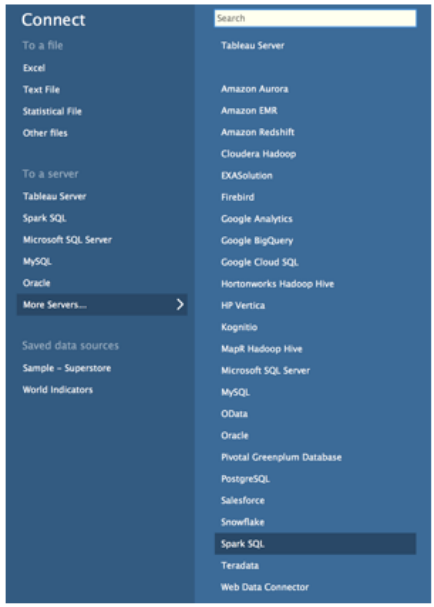
```
/usr/local/Cellar/apache-spark/1.6.2/libexec/sbin/start-thriftserver.sh
```

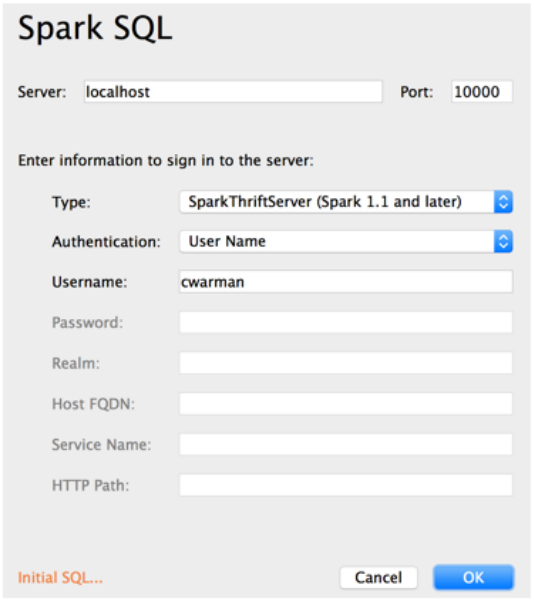
Spark SQL can also act as a distributed query engine
We'll utilize its Thrift JDBC/ODBC server for this.

By default, it listens on port 10,000 and requires a local username (without password) for login
You can lock it down more thoroughly if needed

I referred to this Spark documentation:
https://spark.apache.org/docs/latest/sql-programming-guide.html

Connecting to Thrift Server from Tableau



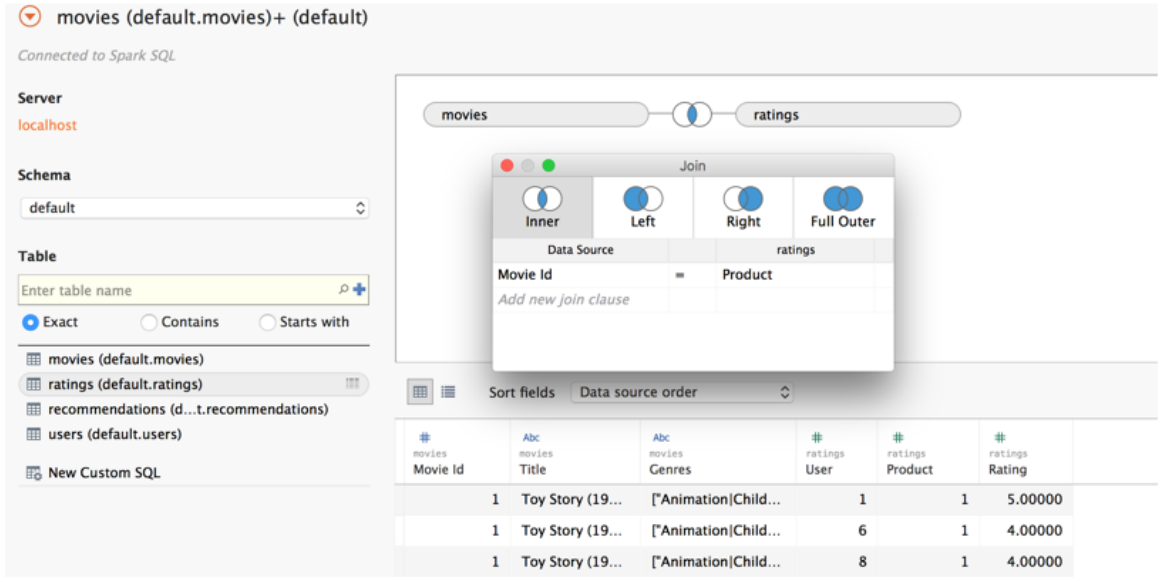


 MAPR 71

Connect from Tableau:
Main "Connect" page > More Servers… > Spark SQL
Server: localhost
Port: 10000
Type: SparkThriftServer (Spark 1.1 and later)
Authentication: User Name
Username: cwarman (notice it's not crwarman)
Select default for the Schema (use magnifying glass "search" functionality)

I referred to this Spark documentation:
https://spark.apache.org/docs/latest/sql-programming-guide.html

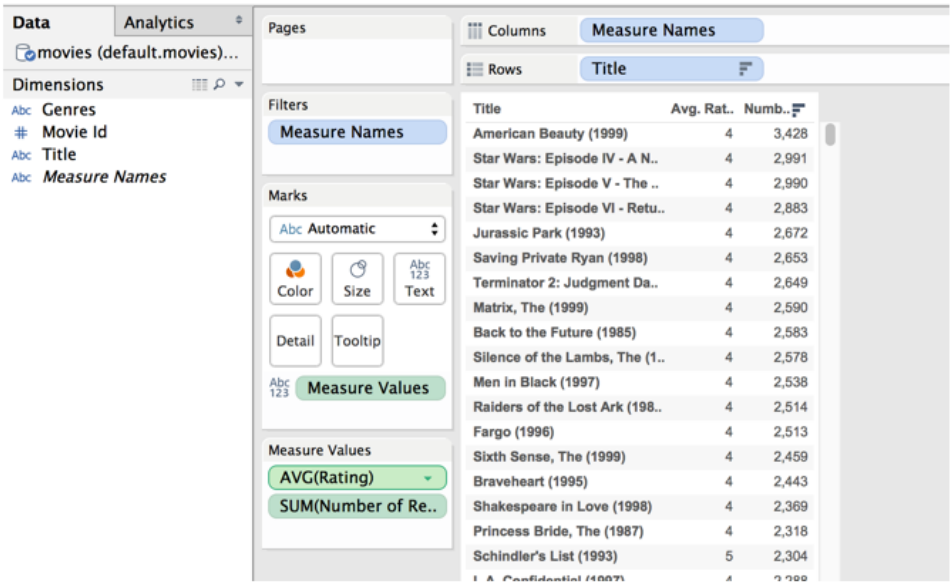
Movie Ratings Data Source



 72

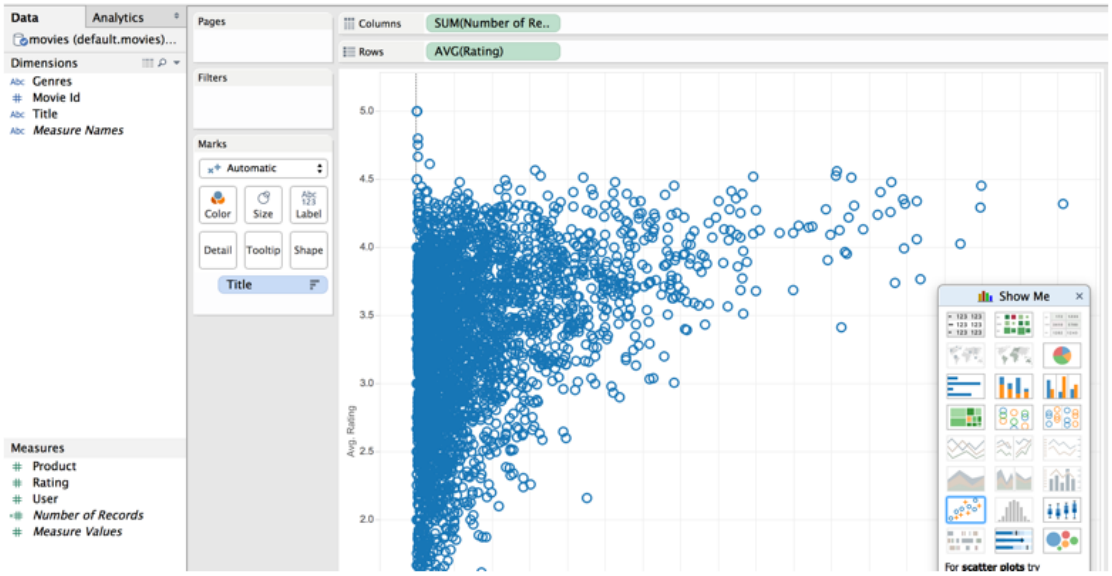
Open **Ratings** workbook if needed

Movie Ratings Workbook



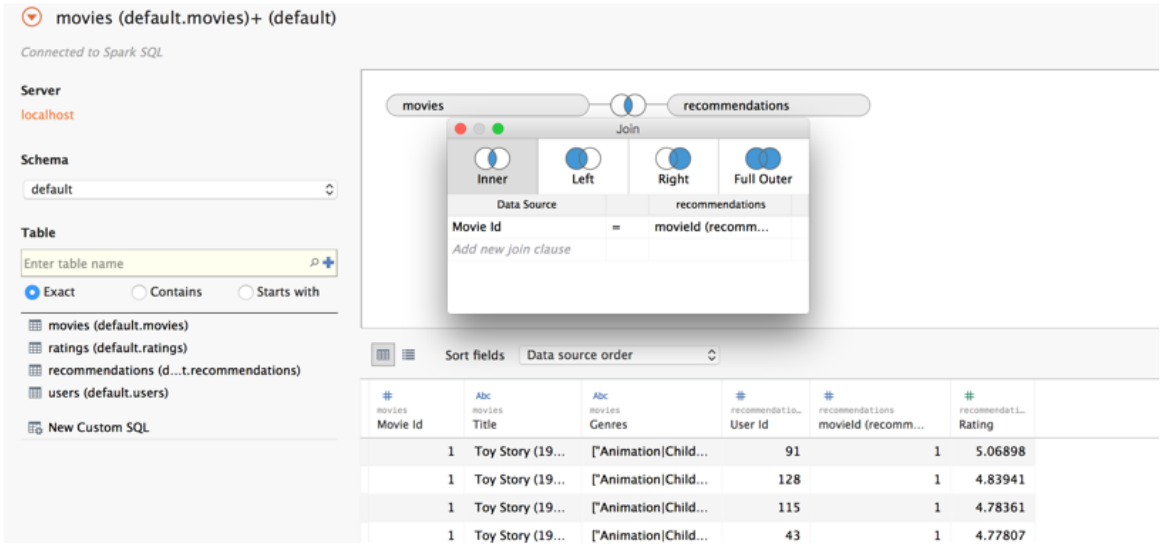
Open **Ratings** workbook if needed

Movie Ratings Workbook

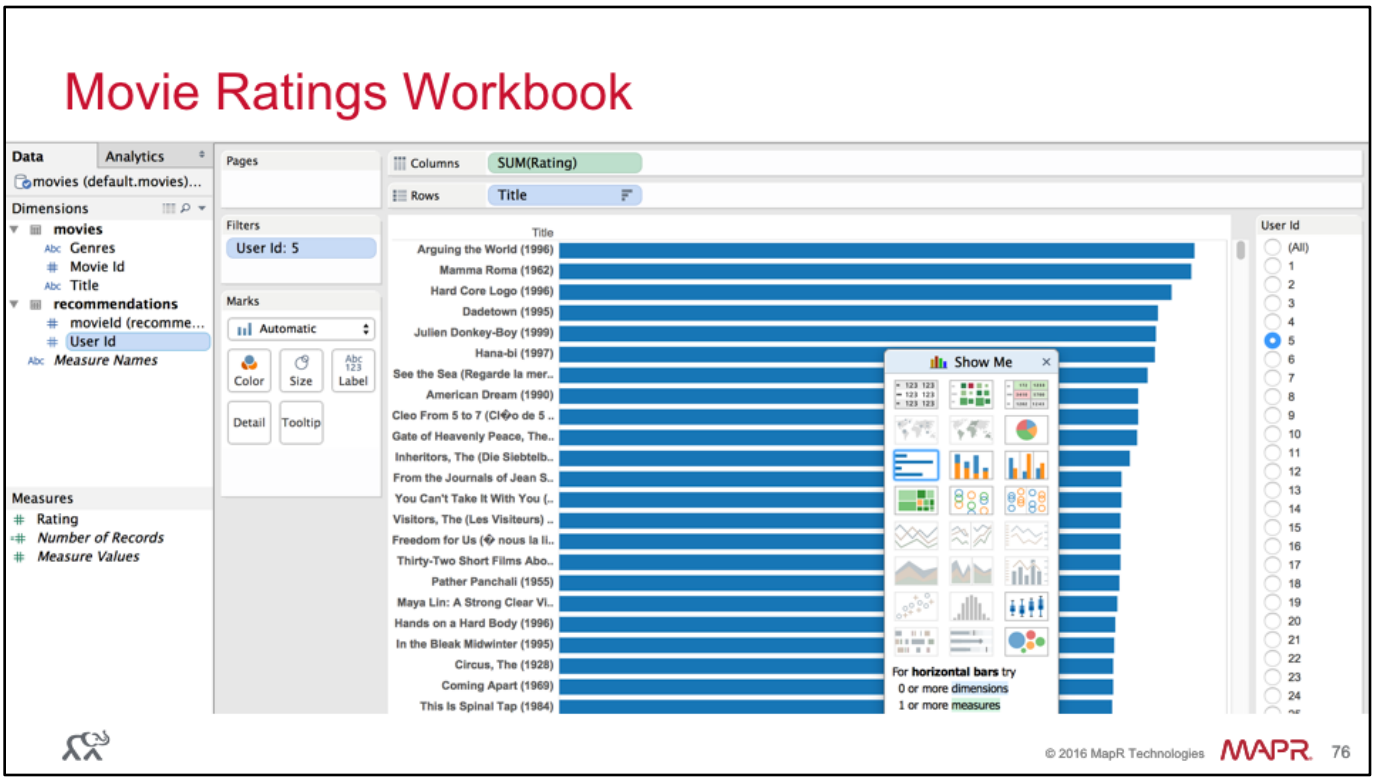


Our own version of a “Magic Quadrant”!

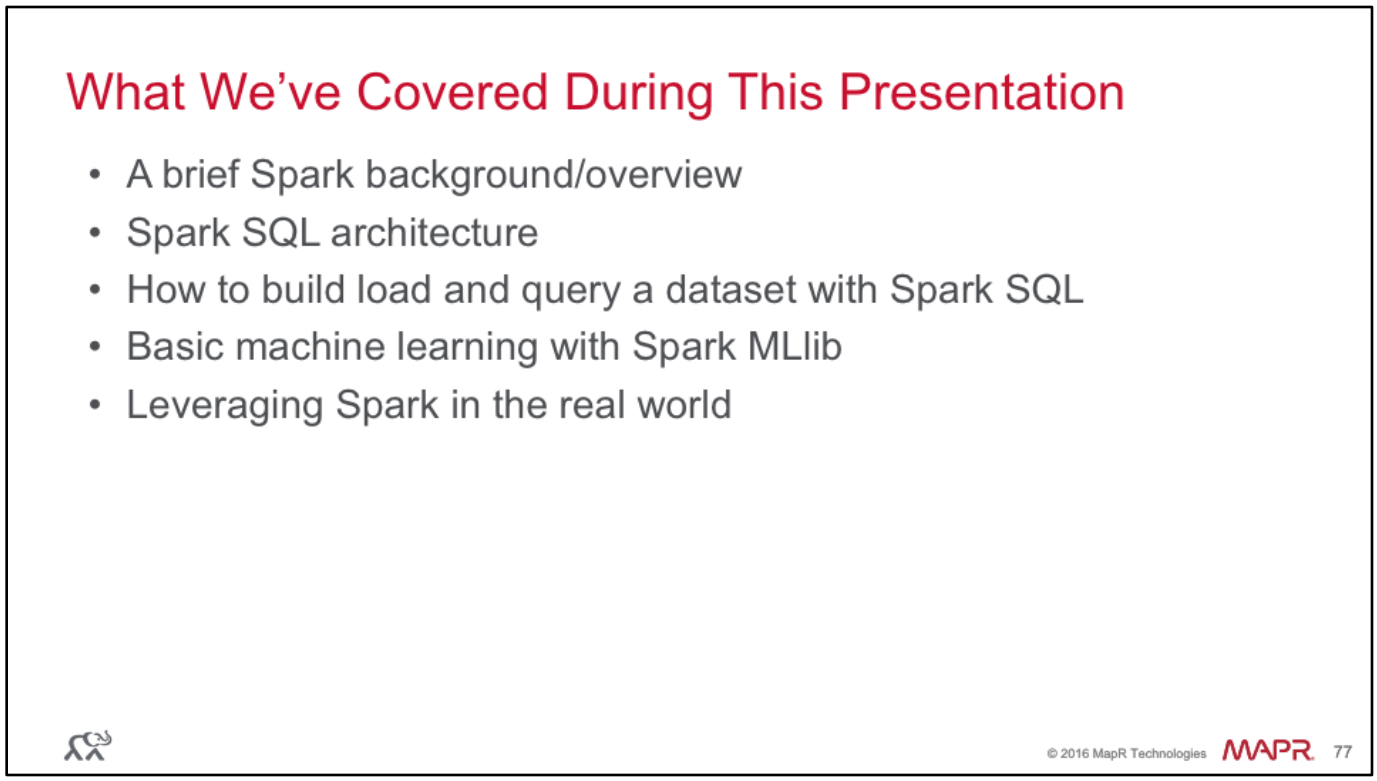
Movie Recommendations Data Source



Open **Recommendations** workbook if needed



Open **Recommendations** workbook if needed



- Spark Background/Overview
  - Brief Spark background
  - Tthe Spark+Hadoop team
  - Spark's five main components
- Spark SQL Architecture
  - Features, Languages, How DataFrames work, The SQLContext, Data sources
- Loading And Querying a Dataset with Spark SQL
  - Live demonstration of setting up a SQLContext
  - Loading it with data
  - Running queries against it
- Machine Learning with Spark MLlib
  - Collaborative filtering basics
  - Alternating Least Squares (ALS) algorithm
  - Live demo of a simple recommender model and training-test loop iterations

- How to connect to Spark SQL using ODBC/JDBC
  - Live demonstration of how to leverage Spark SQL ODBC/JDBC connectivity using Tableau
- Next steps
  - Some Real-World Use Cases
  - Basically answer the questions "What's it good for?" and "Who's using this?"
  - How to download a ready-to-use sandbox VM


Spark SQL & Machine Learning
A Practical Demonstration
Craig Warman
© 2016 MapR Technologies