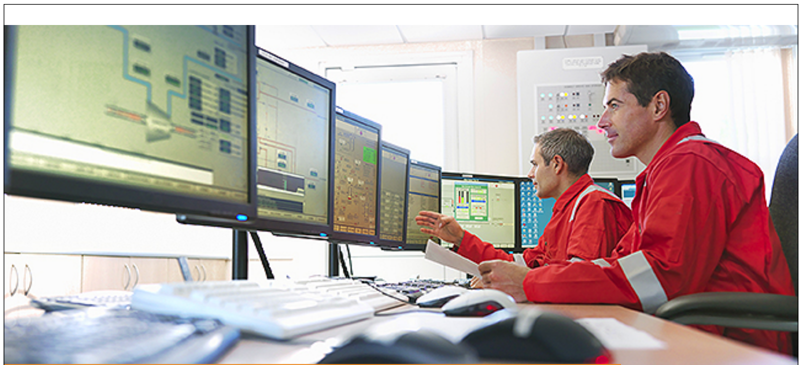
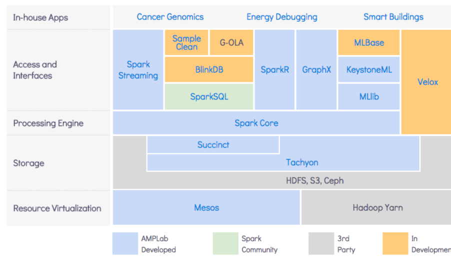


Spark Machine Learning Through Examples

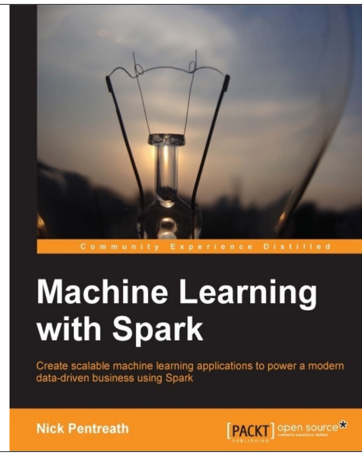
BDAS: the Berkeley Data Analysis Stack



from https://amplab.cs.berkeley.edu/software/

SparkML Overview

- Spark.ml is a uniform set of APIs using data frames to build machine learning pipelines
- Spark.ml is a set of libraries that run on top of Spark
- Unlike packages in Python or R, Spark.ml algorithms run natively in parallel



4

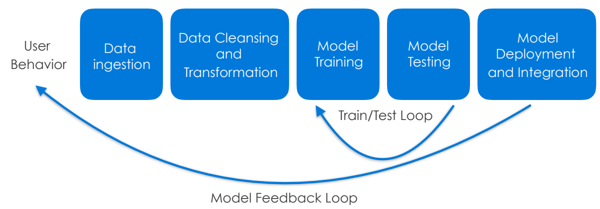
Indispensable. Note that some of the details have changed since the 2nd Edition.

- Official docs
  - https://spark.apache.org/docs/latest/
- Quick Start
  - https://spark.apache.org/docs/latest/quick-start.html
- Programming Guides
  - https://spark.apache.org/docs/latest/programming-guide.html
  - MLlib: https://spark.apache.org/docs/latest/mllib-guide.html
  - Spark SQL: https://spark.apache.org/docs/latest/sql-programming-guide.html
  - GraphX: https://spark.apache.org/docs/latest/graphx-programming-guide.html
  - Streaming: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Agenda

- Machine learning introduction
- Getting started with data
- Supervised learning
  - Creating a recommendation engine
- Other ML examples

A General Machine Learning Pipeline



Machine learning always requires trial and error to determine a good model

General Categories of Machine Learning

- **Supervised learning**
  - An outcome variable guides the learning process
  - Requires a training set with outcome variable already known
  - Model predicts outcome variables for new data
  - Examples: Regressions, Decision trees, Random Forests
- **Unsupervised learning**
  - Outcomes are unknown
  - A self-organizing creates clusters of like samples
  - Human intervention is needed to determine cluster meanings
  - Examples: K-Means clustering, Hidden Markov models
- **Reinforcement learning**
  - Model maximizes a reward function
  - Model can either penalize bad actions, reward good ones, or both
  - Examples: training of self-driving cars based on environment

Agenda

- Machine learning introduction
- Getting started with data
- Supervised learning
  - Creating a recommendation engine
- Other ML examples

Quick exercise

First, we should get a sense of your taste.
Run the following and rate 10 movies:

/rate-ml100k-movies.py

?

We'll Start With Ingestion and Cleansing of the Data

THINKBIG ANALYTICS
A TERADATA COMPANY

User Behavior → Data ingestion → Data Cleansing and Transformation → Model Training → Model Testing → Model Deployment and Integration

Train/Test Loop

Machine learning typically is only as good as the data you feed it

Lab Code is in exercises/SparkML-In-Depth

!

The code we will show on the slides is all from three files in your exercises/SparkML-In-Depth directory

ml-100k-Summary.scala

ml-100k-Data-Frame-Analysis.scala

ml-100k-Rating-Recommendations.scala

All the data files are in hdfs:///data/ml-100k/

Getting Started with MovieLens

- Movielens 100k consists of 100,000 ratings of movies
- Datasets about the movies themselves and the users who did the ratings are also included
- All the data is in text format separated by vertical bars or tabs
- Data origin: http://files.grouplens.org/datasets/movielens/ml-100k.zip
- The ratings file is named u.data as shown below

```
scala> val rating_data = sc.textFile("hdfs:///data/ml-100k/u.data")
rating_data: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[41] at textFile at <console>:21

scala> rating_data.count
res21: Long = 100000

scala> rating_data.take(10)
res22: Array[String] = Array(196	242	3	881250949, 186	302	3	891717742, 22	377	1	878887116, 244	51
	2	880606923, 166	346	1	886397596, 298	474	4	884182806, 115	265	2	881171488, 253	465	5
	891628467, 305	451	3	886324817, 6	86	3	883603013)
```

Movielens users

- The user data is separated by vertical bars
- The fields include an index, age, gender, occupation and zip code

```
scala> val user_data = sc.textFile("hdfs:///data/ml-100k/u.user") // read from HDFS by defaul
user_data: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[49] at textFile at <console>:21

scala> user_data.first
res27: String = 1|24|M|technician|85711

scala> val user_fields = user_data.map(line => line.split("\\|"))
user_fields: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[50] at map at <console>:23

scala> user_fields.take(5)
res28: Array[Array[String]] = Array(Array(1, 24, M, technician, 85711), Array(2, 53, F, other, 94043),
Array(3, 23, M, writer, 32067), Array(4, 24, M, technician, 43537), Array(5, 33, F, other, 15213))
```

Creating Movielens Spark DataFrames

- Case class objects allow us to define schemas
- We then just assign the fields to the schema elements
- The following slide shows how we parse ratings

Creating Movielens Spark DataFrames

```
scala> val rating_data = sc.textFile("hdfs:///data/ml-100k/u.data")
rating_data: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[52] at textFile at <console>:21

scala> case class Rating(userid: Int, itemid: Int, rating: Int, timestamp: String)
defined class Rating

scala> val rating_fields = rating_data.map(line => line.split("\\t"))
rating_fields: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[53] at map at <console>:23

scala> val rating_array = rating_fields.map(p => Rating(p(0).toInt, p(1).toInt, p(2).toInt, p(3)))
rating_array: org.apache.spark.rdd.RDD[Rating] = MapPartitionsRDD[54] at map at <console>:27

scala> val rating_df = rating_array.toDF()
rating_df: org.apache.spark.sql.DataFrame = [userid: int, itemid: int, rating: int, timestamp: string]

scala> rating_df.show(3)
+------+------+------+---------+
|userid|itemid|rating|timestamp|
+------+------+------+---------+
|   196|   242|     3|881250949|
|   186|   302|     3|891717742|
|    22|   377|     1|878887116|
+------+------+------+---------+
only showing top 3 rows
```

Revisiting MovieLens in a more structured way

Let's get started with defining some case classes in the spark-shell

```
scala> // Import Spark SQL data types
scala> import org.apache.spark.sql._
import org.apache.spark.sql._

scala> // Import mllib recommendation data types
scala> import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}
import org.apache.spark.mllib.recommendation.{ALS, MatrixFactorizationModel, Rating}

scala> // input format MovieID::Title::Genres
scala> case class Movie(movieId: Int, title: String)
defined class Movie

scala> // input format is UserID::Gender::Age::Occupation::Zip-code
scala> case class User(userId: Int,  age: Int, gender: String, occupation: String, zip: Int)
defined class User

scala>
```

Revisiting MovieLens in a more structured way

Now we define functions for parsing the various tables

```
scala> // function to parse movie record into Movie class
scala> def parseMovie(str: String): Movie = {
     |       val fields = str.split("\\|")
     |       Movie(fields(0).toInt, fields(1))
     |  }
parseMovie: (str: String)Movie

scala> // function to parse input into User class
scala> def parseUser(str: String): User = {
     |       val fields = str.split("\\|")
     |       assert(fields.size == 5)
     |       User(fields(0).toInt, fields(1).toInt, fields(2).toString, fields(3).toString, fields(4).toInt)
     |  }
parseUser: (str: String)User

scala> // function to parse input UserID|MovieID|Rating
scala> //  Into org.apache.spark.mllib.recommendation.Rating class

scala> def parseRating(str: String): Rating= {
     |       val fields = str.split("\\t")
     |       Rating(fields(0).toInt, fields(1).toInt, fields(2).toDouble)
     | }
parseRating: (str: String)org.apache.spark.mllib.recommendation.Rating
```

Reading in the data

Now read the standard data files along with your ratings of 10 movies

```
// OK, now read in the user and movie info files
val usersDF = sc.textFile("hdfs:///data/ml-100k/u.user").map(parseUser).toDF()
val moviesDF = sc.textFile("hdfs:///data/ml-100k/u.item").map(parseMovie).toDF()
val fileratings = sc.textFile("hdfs:///data/ml-100k/u.data").map(parseRating)
val myratings = sc.textFile("file:///home/hadoop/exercises/SparkML-In-Depth/personalRatings.txt").map(parseRating)
val ratings = fileratings
val ratingsDF = fileratings.toDF()

// register the DataFrames as a temp table for SQL queries
ratingsDF.registerTempTable("ratings")
moviesDF.registerTempTable("movies")
usersDF.registerTempTable("users")
```

Quick exercise

How can we examine the contents of those three dataFrames we just ingested?'

Their names are usersDF, moviesDF, and ratingsDF

Examining the users dataFrame

Let's look at the top 10 entries

```
usersDF.show(10)

+------+---+------+-------------+-----+
|userId|age|gender|   occupation|  zip|
+------+---+------+-------------+-----+
|     1| 24|     M|   technician|85711|
|     2| 53|     F|        other|94043|
|     3| 23|     M|       writer|32067|
|     4| 24|     M|   technician|43537|
|     5| 33|     F|        other|15213|
|     6| 42|     M|    executive|98101|
|     7| 57|     M|administrator|91344|
|     8| 36|     M|administrator| 5201|
|     9| 29|     M|      student| 1002|
|    10| 53|     M|       lawyer|90703|
+------+---+------+-------------+-----+
only showing top 10 rows
```

Examining the users dataFrame with SQL

Of course, one of the features of dataFrames is that SQL works too

```
sqlContext.sql("SELECT * FROM users LIMIT 10").show
+------+---+------+-------------+-----+
|userId|age|gender|   occupation|  zip|
+------+---+------+-------------+-----+
|     1| 24|     M|   technician|85711|
|     2| 53|     F|        other|94043|
|     3| 23|     M|       writer|32067|
|     4| 24|     M|   technician|43537|
|     5| 33|     F|        other|15213|
|     6| 42|     M|    executive|98101|
|     7| 57|     M|administrator|91344|
|     8| 36|     M|administrator| 5201|
|     9| 29|     M|      student| 1002|
|    10| 53|     M|       lawyer|90703|
+------+---+------+-------------+-----+
```

Examining the movies dataFrame

Again, the top 10 entries

```
moviesDF.show(10)

+-------+--------------------+
|movieId|               title|
+-------+--------------------+
|      1|    Toy Story (1995)|
|      2|    GoldenEye (1995)|
|      3|   Four Rooms (1995)|
|      4|   Get Shorty (1995)|
|      5|      Copycat (1995)|
|      6|Shanghai Triad (Y...|
|      7|Twelve Monkeys (1...|
|      8|         Babe (1995)|
|      9|Dead Man Walking ...|
|     10|  Richard III (1995)|
+-------+--------------------+
only showing top 10 rows
```

Examining the ratings dataFrame

And finally the ratings

```
ratingsDF.show(10)

+----+-------+------+
|user|product|rating|
+----+-------+------+
| 196|    242|   3.0|
| 186|    302|   3.0|
|  22|    377|   1.0|
| 244|     51|   2.0|
| 166|    346|   1.0|
| 298|    474|   4.0|
| 115|    265|   2.0|
| 253|    465|   5.0|
| 305|    451|   3.0|
|   6|     86|   3.0|
+----+-------+------+
only showing top 10 rows
```

Quick exercise

How can we find the ratings for Star Wars (1977)?

?

Star Wars Ratings Answer

Let's do some simple queries both without and with SQL

Let's find Star Wars in the movies dataFrame

```
scala> moviesDF.filter(moviesDF("title").contains("Star Wars")).show
+-------+----------------+
|movieId|           title|
+-------+----------------+
|     50|Star Wars (1977)|
+-------+----------------+

// You could also do this by saying

scala> sqlContext.sql("SELECT movieID, title FROM movies WHERE title LIKE \"Star Wars%\"").show
+-------+----------------+
|movieID|           title|
+-------+----------------+
|     50|Star Wars (1977)|
+-------+----------------+
```

Denormalizing the data using joins

We could just search for movie ID 50, but most people like to see titles

Let's join together ratings and movie titles so we can search for the title Star Wars

```
scala> val ratings_with_titleDF = ratingsDF.join(moviesDF, ratingsDF("product") === moviesDF("movieID"))
ratings_with_titleDF: org.apache.spark.sql.DataFrame = [user: int, product: int, rating: double, movieId: int,
title: string]

scala> ratings_with_titleDF.show(5)
+----+-------+------+-------+-------------------+
|user|product|rating|movieId|              title|
+----+-------+------+-------+-------------------+
| 109|     31|   4.0|     31|Crimson Tide (1995)|
| 144|     31|   3.0|     31|Crimson Tide (1995)|
|  90|     31|   4.0|     31|Crimson Tide (1995)|
| 244|     31|   4.0|     31|Crimson Tide (1995)|
| 313|     31|   4.0|     31|Crimson Tide (1995)|
+----+-------+------+-------+-------------------+
only showing top 5 rows
```

Denormalizing the data using joins

Let's find Star Wars in the denormalized dataFrame

```
// See ratings for Star Wars now

sscala> val starwars = ratings_with_titleDF.filter($"title").contains("Star Wars"))
starwars: org.apache.spark.sql.DataFrame = [user: int, product: int, rating: double, movieId: int, title: string]

scala> starwars.show(5)
+----+-------+------+-------+----------------+
|user|product|rating|movieId|           title|
+----+-------+------+-------+----------------+
| 290|     50|   5.0|     50|Star Wars (1977)|
|  79|     50|   4.0|     50|Star Wars (1977)|
|   2|     50|   5.0|     50|Star Wars (1977)|
|   8|     50|   5.0|     50|Star Wars (1977)|
| 274|     50|   5.0|     50|Star Wars (1977)|
+----+-------+------+-------+----------------+
only showing top 5 rows
```

This $"title" is shorthand for

`ratings_with_titleDF("title")`

Quick exercise

On average, did men or women rate Star Wars higher?

Write a query that finds the average rating for Star Wars grouped by the field gender

?

Some quick queries

To understand who is doing the rating, let's also merge in the user database

```
scala> val denorm_ratingsDF = ratings_with_titleDF.
                join(usersDF, ratings_with_titleDF("user") === usersDF("userId"))

scala> val starwars = denorm_ratingsDF.filter(denorm_ratingsDF("title").
                contains("Star Wars"))

scala> starwars.show(5)
+----+-------+------+-------+----------------+------+---+------+-------------+-----+
|user|product|rating|movieId|           title|userId|age|gender|   occupation|  zip|
+----+-------+------+-------+----------------+------+---+------+-------------+-----+
| 231|     50|   4.0|     50|Star Wars (1977)|   231| 48|     M|    librarian|01080|
| 831|     50|   5.0|     50|Star Wars (1977)|   831| 21|     M|        other|33765|
|  32|     50|   4.0|     50|Star Wars (1977)|    32| 28|     F|      student|78741|
| 232|     50|   4.0|     50|Star Wars (1977)|   232| 45|     M|    scientist|99709|
| 432|     50|   5.0|     50|Star Wars (1977)|   432| 22|     M|entertainment|50311|
+----+-------+------+-------+----------------+------+---+------+-------------+-----+
only showing top 5 rows
```

Average ratings of Star Wars by gender

Now group by gender and take the average rating

```
scala> starwars.persist
scala> starwars.groupBy("gender").agg(avg($"rating")).show
+------+-----------------+
|gender|      avg(rating)|
+------+-----------------+
|     F|4.245033112582782|
|     M|4.398148148148148|
+------+-----------------+
```

Average ratings of Star Wars by gender

Another approach is to use SQL

```
scala> starwars.registerTempTable("starwars")
scala> sqlContext.sql("SELECT gender, AVG(rating) FROM starwars GROUP BY gender").show
+------+-----------------+
|gender|              _c1|
+------+-----------------+
|     F|4.245033112582782|
|     M|4.398148148148148|
+------+-----------------+
```

Average ratings of Star Wars by gender

And perhaps we should check this against the total set of movies for sanity. I'd expect it to be higher than average.

```
scala> starwars.registerTempTable("starwars")
scala> sqlContext.sql("SELECT gender, AVG(rating) FROM starwars GROUP BY gender").show
+------+-----------------+
|gender|              _c1|
+------+-----------------+
|     F|4.245033112582782|
|     M|4.398148148148148|
+------+-----------------+

// check the results against the total set of movies for sanity
scala> denorm_ratingsDF.groupBy("gender").agg(avg($"rating")).show // compare with total set
+------+------------------+
|gender|       avg(rating)|
+------+------------------+
|     F|3.5315073815073816|
|     M|3.5292889846485322|
+------+------------------+
```

Some quick SQL queries

Let's look at some stats on the overall ratings

One-line query string allows us to paste this into spark-shell

```
// Get the max, min ratings along with the count of users who have rated a movie.

scala> val results =sqlContext.sql("SELECT movies.title, movierates.maxr, movierates.minr, movierates.cntu FROM
(SELECT ratings.product, max(ratings.rating) AS maxr, min(ratings.rating) AS minr, COUNT(DISTINCT user) AS cntu
FROM ratings group BY ratings.product ) movierates JOIN movies ON movierates.product=movies.movieId ORDER BY
movierates.cntu DESC")
scala> results.show(5)
+--------------------+----+----+----+
|               title|maxr|minr|cntu|
+--------------------+----+----+----+
|    Star Wars (1977)| 5.0| 1.0| 583|
|      Contact (1997)| 5.0| 1.0| 509|
|        Fargo (1996)| 5.0| 1.0| 508|
|Return of the Jed...| 5.0| 1.0| 507|
|    Liar Liar (1997)| 5.0| 1.0| 485|
+--------------------+----+----+----+
only showing top 5 rows
```

Some quick SQL queries

Show the top 5 most-active users and how many times they rated a movie

```
// Show the top 5 most-active users and how many times they rated a movie

scala> val mostActiveUsersSchemaRDD = sqlContext.sql("SELECT ratings.user, count(*) AS ct FROM ratings GROUP BY
ratings.user ORDER BY ct DESC LIMIT 5")

scala> println(mostActiveUsersSchemaRDD.collect().mkString("\n"))
[405,737]
[655,685]
[13,636]
[450,540]
[276,518]
```

Some quick SQL queries

Show the top 5 most-active users and how many times they rated a movie

```
scala> // Find the movies that user 92 rated higher than 4

scala> val results =sqlContext.sql("SELECT ratings.user, ratings.product, ratings.rating, movies.title FROM
ratings JOIN movies ON movies.movieId=ratings.product WHERE ratings.user=92 AND ratings.rating > 4")
results: org.apache.spark.sql.DataFrame = [user: int, product: int, rating: double, title: string]

scala> results.show(5)
+----+-------+------+--------------------+
|user|product|rating|               title|
+----+-------+------+--------------------+
|  92|    433|   5.0|     Heathers (1989)|
|  92|    238|   5.0|Raising Arizona (...|
|  92|    640|   5.0|Cook the Thief Hi...|
|  92|     50|   5.0|    Star Wars (1977)|
|  92|    855|   5.0|         Diva (1981)|
+----+-------+------+--------------------+
only showing top 5 rows
```

Agenda

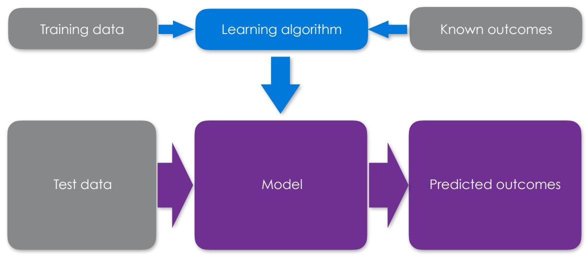
- Machine learning introduction
- Getting started with data
- Supervised learning
  - Creating a recommendation engine
- Other ML examples

Defining a recommendation engine

- Examples
  - Amazon's "Recommended for you" service
  - Netflix's "Top picks for…"
- Most effective when the use case has
  - A large number of available options
  - A significant degree of personal taste is involved

A supervised learning method that generalizes a small sample of ratings or votes onto a much larger set of objects

Recommendation engine data flow



Our algorithm: collaborative filtering using matrix factorization

- One of many supervised recommendation engines
- Uses not only your past ratings, but ratings of other users of similar content (i.e., it relies on crowdsourcing ratings of new materials)

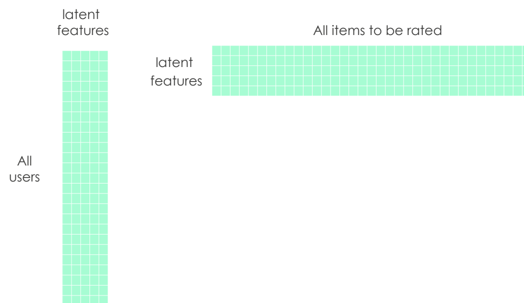
	Batman	Star Wars	Titanic
Bill	3	3	
Jane		2	4
Tom		5	

Our problem: the full ratings matrix is very sparse

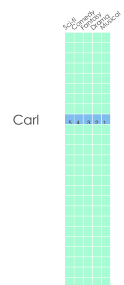
- Most users only rate a few items
- Some users don't rate any
- Imagine 1 million users and 1 million items: the matrix has 1 trillion entries!

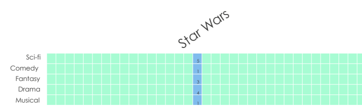
What we want: break down these ratings into two factor matrices of other features

- Latent features don't necessarily need to have meaning
- For illustration, imagine that latent features could be demographics, genres, or actors



A too simple example: Will Carl like Star Wars?





- Carl usual rating for Sci-Fi is a 5
- Viewer consensus is that Star Wars is a Sci-Fi movie, a Fantasy and a Drama
- Combining these ratings will forecast a rating for Carl's likely rating of Star Wars
- The challenge is deriving these latent features and matrices

First step: let's divide our dataset into a training set and a test set

We will also include our personal ratings we did under user number 0

```
scala> val splits = ratings.randomSplit(Array(0.8, 0.2), 0L)
scala> val trainingRDD = splits(0).cache()
scala> val testRatingsRDD = splits(1).cache()

// now add our ratings to the training set

scala> val trainingRatingsRDD = trainingRDD.union(myratings).cache()
scala> val numTraining = trainingRatingsRDD.count()
scala> val numTest = testRatingsRDD.count()
scala> println(s"Training: $numTraining, test: $numTest.")

Training: 79838, test: 20173.
```

Now we train up the Alternating Least Squares module on our training set

Three arguments

1. rank: number of hidden features to use in approximation matrices
2. iterations: number of times to run the model and approximate a better result
3. lambda: tunable parameter for regularization and fitting

```
scala> val rank = 10        // number of hidden features in approximation matrices
scala> val iterations = 10  // iterations of model to run
scala> val lambda = 0.01    // tunable parameter controlling regularization and fitting

scala> val model = ALS.train(trainingRatingsRDD, rank, iterations, lambda)
```

What can we do with our MatrixFactorizationModel?

It's a recommendation engine, so maybe we can get some movie recommendations for you

```
scala> val movieTitles=moviesDF.map(array => (array(0), array(1))).collectAsMap()
// We'll test movieTitles by using it on our ratings (user 0)

scala> val user0ratings = trainingRatingsRDD.filter(rating => rating.user == 0)
scala> user0ratings.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
(Toy Story (1995),5.0)
(Independence Day (ID4) (1996),4.0)
(Dances with Wolves (1990),5.0)
(Star Wars (1977),5.0)
(Mission: Impossible (1996),2.0)
(Ace Ventura: Pet Detective (1994),2.0)
(Die Hard: With a Vengeance (1995),1.0)
(Batman Forever (1995),2.0)
(Pretty Woman (1990),5.0)
(Men in Black (1997),3.0)
(Dumb & Dumber (1994),1.0)

scala> val topRecForUser0 = model.recommendProducts(0, 3)
scala> topRecForUser0.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
Gone Fishin' (1997),9.254525125806898)
(Little Rascals, The (1994),8.424405239923589)
(Robert A. Heinlein's The Puppet Masters (1994),8.143378514407349)
```

Let's check to see if other users get different results

Try user 1

```
scala> val user1ratings = trainingRatingsRDD.filter(_.user == 1)
scala> user1ratings.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
scala> val topRecForUser1 = model.recommendProducts(1, 3)
scala> topRecForUser1.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)

scala> topRecForUser1.map(rating => (movieTitles(rating.product), rating.rating)).foreach(println)
(Miserables, Les (1995),6.684018155098329)
(Traveller (1997),6.662542798739833)
(Chungking Express (1994),6.17771009064068)
```

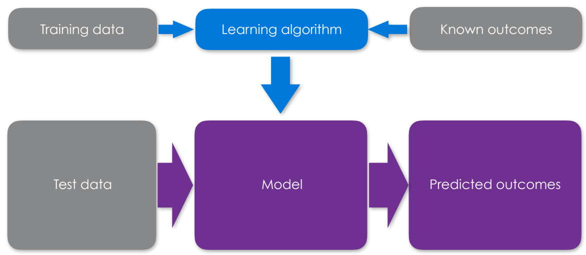
Question

Are these predictions any good?

Let's try doing predictions on the test set and compare them with actual results

?

Recommendation engine data flow



Get predicted ratings for the test dataset

We'll define a helper case class to pull out just the user and product for the predict method; it wants a tuple as an input, not a triple

```
// First, define a helper case class to
// get user product pair from testRatingsRDD
// generates a simple pair of just user and product

scala> val testUserProductRDD = testRatingsRDD.map {
     |   case Rating(user, product, rating) => (user, product)
     | }

scala> // get predicted ratings to compare to test ratings
scala> // remember that testRatingsRDD is our test sample of all our data
scala> val predictionsForTestRDD  = model.predict(testUserProductRDD)
scala> predictionsForTestRDD.take(5).mkString("\n")

res203: String =
Rating(634,1084,4.408128796215626)
Rating(194,1410,2.997288400231771)
Rating(833,667,2.2414365326076524)
Rating(13,667,0.4987697452089243)
Rating(201,667,3.5635611911634286)
```

Now transform the user-product tuple into a key

We'll define a helper case class to pull out just the user and product for the predict method; it wants a tuple, not a triple as n input

```
// prepare  predictions for comparison
scala> val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
     |   case Rating(user, product, rating) => ((user, product), rating)
     | }

// prepare the original test data for comparison
scala> val testKeyedByUserProductRDD = testRatingsRDD.map{
     |   case Rating(user, product, rating) => ((user, product), rating)
     | }

//Join the  test with  predictions; (user, product) is the key to join on
scala> val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD.join(predictionsKeyedByUserProductRDD)

// print the (user, product),(test rating, predicted rating)
scala> testAndPredictionsJoinedRDD.take(3).mkString("\n")
res52: String =
((115,218),(3.0,3.364017989843915))
((608,729),(4.0,4.896983761195552))
((151,605),(4.0,3.195194072740393))
```

Check for false positives

We're checking for an actual test data set rating which is 1 or 0 and a predicted rating that is 4 or 5. That's clearly a bad prediction

```
scala> val falsePositives =(testAndPredictionsJoinedRDD.filter{
     |   case ((user, product), (ratingT, ratingP)) =>
     |         (ratingT <= 1 && ratingP >=4)
     |   })

scala> falsePositives.take(3)
res57: Array[((Int, Int), (Double, Double))] = Array(((326,481),(1.0,4.269430966364969)),
((551,824),(1.0,5.034163966959284)), ((433,1598),(1.0,5.081932999327656)))

scala> falsePositives.count
res58: Long = 108
```

Check the Mean Absolute Error

Mean Absolute Error (MAE) just calculates the absolute value of the difference between the test rating and the predicted rating. We'll calculate the mean of all the test MAEs to measure how good our model is

```
//Evaluate the model using Mean Absolute Error (MAE) between test and predictions

scala> val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
     |   case ((user, product), (testRating, predRating)) =>
     |     val err = (testRating - predRating)
     |     Math.abs(err)
     | }.mean()
meanAbsoluteError: Double = 0.8236981707203458
```

Exercise

Can we improve this? What if we did more interations? More factors? Different lambdas?

Manipulate the model parameters and see who can create the best Mean Absolute Error on the test data set.

?

Model and MAE calculations

```
// Set up the parameters for training

val rank = 10            // number of hidden features in approximation matrices
val iterations = 10      // iterations of model to run
val lambda = 0.01        // tunable parameter controlling regularization and fitting
val model = ALS.train(trainingRatingsRDD, rank, iterations, lambda)

val testUserProductRDD = testRatingsRDD.map {
  case Rating(user, product, rating) => (user, product)
}

val predictionsForTestRDD  = model.predict(testUserProductRDD)
val predictionsKeyedByUserProductRDD = predictionsForTestRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}
val testKeyedByUserProductRDD = testRatingsRDD.map{
  case Rating(user, product, rating) => ((user, product), rating)
}
val testAndPredictionsJoinedRDD = testKeyedByUserProductRDD.join(predictionsKeyedByUserProductRDD)

//Evaluate the model using Mean Absolute Error (MAE) between test and predictions
val meanAbsoluteError = testAndPredictionsJoinedRDD.map {
  case ((user, product), (testRating, predRating)) =>
    val err = (testRating - predRating)
    Math.abs(err)
}.mean()
```

Bottom Line: Maybe We Could Do Better?

- Our results are OK but not amazing.
- On the other hand, we only wrote a couple pages of code and did a little exploration of the space.
- We aren't sure if we are good or just lucky so we might want to do some resampling or boosting to understand it better

Agenda

- Machine learning introduction
- Getting started with data
- Supervised learning
  - Creating a recommendation engine
- Other ML examples

SparkML Has Many Machine Learning Functions

- Basic statistics
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

All The SparkML Tools Follow A Similar Pattern

- Get your data into the right types of objects
- Generate a model over a training set
- Apply that model to your test set using "predict"
- Evaluate the results and iterate

SparkML does all the hard work of figuring out how to run those algorithms in parallel.

Example: k-means (exercises/SparkML/KMeansClustering.scala)

```
// Load and parse the data

scala> val data = sc.textFile("/data/sparkml-data/kmeans_data.txt")
scala> val parsedData = data.map(s => Vectors.dense(s.split(' ').map(_.toDouble))).cache()

scala> // Cluster the data into two classes using KMeans

scala> val numClusters = 2
scala> val numIterations = 20
scala> val clusters = KMeans.train(parsedData, numClusters, numIterations)
16/04/17 19:11:06 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
16/04/17 19:11:06 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS

// Evaluate clustering by computing Within Set Sum of Squared Errors

scala> val WSSSE = clusters.computeCost(parsedData)
WSSSE: Double = 0.11999999999994547
scala> println("Within Set Sum of Squared Errors = " + WSSSE)
Within Set Sum of Squared Errors = 0.11999999999994547

scala>
```

Example: k-means data

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

```
val numClusters = 2
println("Within Set Sum of Squared Errors = " + WSSSE)
Within Set Sum of Squared Errors = 0.11999999999994547
```

```
val numClusters = 3
println("Within Set Sum of Squared Errors = " + WSSSE)
Within Set Sum of Squared Errors = 0.07499999999994544
```

Example: RandomForest Classification

Code is in exercises/SparkML-In-Depth/RandomForestClassification.scala

```scala
// The example below demonstrates how to load a LIBSVM data file, parse
// it as an RDD of LabeledPoint and then perform classification using a
// Random Forest. The test error is calculated to measure the algorithm
// accuracy.

import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.model.RandomForestModel
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "/data/sparkml-data/sample_libsvm_data.txt")
// Split the data into training and test sets (30% held out for testing)
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a RandomForest model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val numTrees = 3 // Use more in practice.
val featureSubsetStrategy = "auto" // Let the algorithm choose.
val impurity = "gini"
val maxDepth = 4
val maxBins = 32

val model = RandomForest.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
  numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble / testData.count()
println("Test Error = " + testErr)
println("Learned classification forest model:\n" + model.toDebugString)

// Save and load model
model.save(sc, "myModelPath")
val sameModel = RandomForestModel.load(sc, "myModelPath")
```

Example: RandomForest

SVM input data is at “hdfs:///data/sparkml-data/sample_libsvm_data.txt”

Looks like this….

0 128:51 129:159 130:253 131:159 132:50 155:48 156:238 …

Output

```
Test Error =
0.08823529411764706
Tree 0:
  If (feature 517 <= 41.0)
   Predict: 0.0
  Else (feature 517 > 41.0)
   If (feature 371 <= 7.0)
    Predict: 1.0
   Else (feature 371 > 7.0)
    Predict: 0.0
Tree 1:
  If (feature 378 <= 0.0)
   Predict: 0.0
  Else (feature 378 > 0.0)
   If (feature 522 <= 21.0)
    Predict: 1.0
   Else (feature 522 > 21.0)
    Predict: 0.0
```

```
Tree 2:
    If (feature 540 <= 0.0)
     If (feature 235 <= 0.0)
      Predict: 1.0
     Else (feature 235 > 0.0)
      Predict: 0.0
    Else (feature 540 > 0.0)
     If (feature 626 <= 0.0)
      Predict: 1.0
     Else (feature 626 > 0.0)
      Predict: 0.0
```

Other examples in exercises/SparkML

- ClassificationTrees.scala
- GradientBoostedClassification.scala
- GradientBoostedRegression.scala
- KMeansClustering.scala
- NaiveBayes.scala
- RandomForestClassification.scala
- RandomForestRegression.scala
- RegressionTrees.scala

Be sure to pull the versions from /usr/lib/spark/examples for the version of Spark you are running

Summary

- Machine learning in Spark is built-into the platform
- The biggest challenges in machine learning are
  - Getting data into the right form
  - Understanding the output
  - Optimizing the model
- Despite the challenges, big data machine learning is probably easier to implement in Spark than on any other platform