CS535 BIG DATA

PART 1. BATCH COMPUTING MODELS FOR BIG DATA ANALYTICS

2. LARGE SCALE DATA ANALYSIS USING SPARK WITH CASE STUDY

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

- Sign up sheet for PA1 is available
- Term Project
  - Google computing cluster credit is available
  - Optional

Objectives

- Large scale data analysis using Spark with case study
  - Decision tree/Random Forest
- Recommendation systems
  - Collaborative filtering
  - Latent factor approach

Large scale data analysis using Spark with case study

Predicting Forest Cover with Decision Trees

Tuning Decision Trees

- Spark tries a number of combinations of impurity measure, maximum depth or number of bins and reports the results

```
val evaluations =
   for (impurity <- Array("gini", "entropy");
        depth <- Array(1, 20);
        bins <- Array(10, 300))
     yield {
       val model = DecisionTree.trainClassifier(trainData, 7,
             Map[Int, Int](), impurity, depth, bins)
       val predictionsAndLabels = cvData.map(example = >
         (model.predict( example.features), example.label) )
       val accuracy =
          new MulticlassMetrics(predictionsAndLabels).
          precision ((impurity, depth, bins), accuracy)
      }
      evaluations.sortBy(_._2). reverse.foreach( println) …
```


Tuning Decision Trees

- continued

```
(( entropy, 20,300), 0.9125545571245186)
(( gini, 20,300), 0.9042533162173727)
(( gini, 20,10), 0.8854428754813863)
(( entropy, 20,10), 0.8848951647411211)
(( gini, 1,300), 0.6358065896448438)
(( gini, 1,10), 0.6355669661959777)
(( entropy, 1,300), 0.4861446298673513)
(( entropy, 1,10), 0.4861446298673513)
```

Categorical Features Revisited

- `Map[Int, Int]()`
  - Keys
    - **Indices of features** in the input Vector
  - Values
    - Distinct **value counts**
- Empty `Map()`
  - No features should be treated as categorical
  - All are numeric
- Numeric representation of categorical features
  - It can cause errors
  - The algorithm would be trying to learn from an ordering that has no meaning

Treating the categorical features with **one-hot** encoding

- Encodes the categorical features as several binary 0/1 values
- Any decision rule on the "numeric" features will choose thresholds between 0 and 1
  - All are equivalent since all values are 0 or 1
- Considers the values of the underlying categorical feature **individually**
  - Increases memory usage

Converting one-hot encoding to 1-n encoding [1/3]

```
val data = rawData.map { line = >
  val values = line.split(','). map(_.toDouble)
  val wilderness = values.slice(10, 14).indexOf(1.0). toDouble
  val soil = values.slice(14, 54).indexOf(1.0).toDouble
  val featureVector = Vectors.dense(values.slice(0, 10) :+
wilderness :+ soil)
  val label = values.last - 1
  LabeledPoint( label, featureVector)
}
```

- 4 "wilderness" features
- 40 "soil" features
- Add derived features back to first 10

Converting one-hot encoding to 1-n encoding [2/3]

```
val evaluations =
  for (impurity <- Array("gini", "entropy"); depth <-
Array( 10, 20, 30); bins <- Array(40, 300))
  yield {
     val model =
   DecisionTree.trainClassifier(trainData,7,Map(10->4,11->40),
impurity, depth, bins)
     val trainAccuracy = getMetrics(model, trainData).
precision val cvAccuracy = getMetrics( model,cvData).
precision ((impurity, depth, bins),(trainAccuracy,cvAccuracy))
}
```

- Specify value count for categorical features 10, 11
  - Causes these features to be treated as categorical

Converting one-hot encoding to 1-n encoding [3/3]

```
(( entropy, 30,300),( 0.9996922984231909,0.9438383977425239))
(( entropy, 30,40),( 0.9994469978654548,0.938934581368939))
(( gini, 30,300),( 0.9998622874061833,0.937127912178671))
(( gini, 30,40),( 0.9995180059216415,0.9329467634811934))
(( entropy, 20,40),( 0.9725865867933623,0.9280773598540899))
(( gini, 20,300),( 0.9702347139020864,0.9249630062975326))
(( entropy, 20,300),( 0.9643948392205467,0.9231391307340239))
(( gini, 20,40),( 0.9679344832334917,0.9223820503114354))
(( gini, 10,300),( 0.7953203539213661,0.7946763481193434))
(( gini, 10,40),( 0.7880624698753701,0.7860215423792973))
…
```

- Tree-building process completes several times **faster**
- By treating categorical features as categorical features, it **improves accuracy by almost 3%**

Does decision tree algorithm build the same tree every time?

- Over N values
  - There are 2^N-2 possible decision rules
- Decision trees use several heuristics to narrow down the rules to be considered
  - The process of picking rules involves some randomness
  - Only a few features, picked at random, are looked at each time
  - Only values from a random subset of the training data are looked
  - Trades a bit of **accuracy** for a lot of **speed**
- Decision tree algorithm **won't build the same tree every time**

RandomForest

```
val forest = RandomForest.trainClassifier(
       trainData, 7, Map( 10 -> 4, 11 -> 40), 20,
       "auto", "entropy", 30, 300)
```

- Number of trees to build
  - Here 20
- "auto"
  - The strategy for choosing which features to evaluate at each level of the tree
  - The random decision forest implementation will NOT even consider every feature as the basis of a decision rule
    - Only a subset of all features

Making predictions

- The results of the `DecisionTree` and `RandomForest` training
  - `DecisionTreeModel` and `RandomForestModel` objects
  - `predict()` method
    - Accepts a Vector object
  - We can classify a new example by converting it to a feature vector in the same way and predicting its target class

```
val input = "2709,125,28,67,23,3224,253,207,61,6094,0,29"
val vector = Vectors.dense(input.split(',').map(_.toDouble)
)
forest.predict(vector)
```


Large scale data analysis using Spark
CASE STUDY: Recommending Music and the Audioscrobbler Data Set

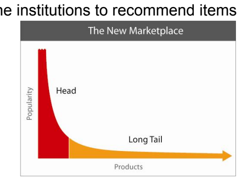
This material is built based on

- Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (August 2009), 30-37. DOI=10.1109/MC.2009.263 http://dx.doi.org/10.1109/MC.2009.263
- Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining* (ICDM '08). IEEE Computer Society, Washington, DC, USA, 263-272. DOI=http://dx.doi.org/10.1109/ICDM.2008.22
- Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills, Advanced Analytics with Spark, O'Reilly, 2015

"What percentage of the top 10,000 titles in any online media store (Netflix, iTunes, Amazon, or any other) will rent or sell at least once a month?"

The long tail phenomenon [1/2]

- Distribution of numbers with a portion that has a large number of occurrences far from the "head" or central part of the distribution
  - The vertical axis represents popularity
  - The items are ordered on the horizontal axis according to their popularity
  - The long-tail phenomenon forces online institutions to recommend items to individual users



Erik Brynjolfsson, Yu (Jeffrey) Hu, and Duncan Simester. 2011. Goodbye Pareto Principle, Hello Long Tail: The Effect of Search Costs on the Concentration of Product Sales. *Manage. Sci.* 57, 8 (August 2011), 1373-1386. DOI=http://dx.doi.org/10.1287/mnsc.1110.1371

The long tail phenomenon [2/2]

- "Touching the Void", Joi Simpson, 1988

- "Into Thin Air: A Personal Account of the Mt. Everest Disaster" , Jon Krakauer, 1997

Recommendation systems

- Seek to predict the "rating" or "preference" that a user would give to an item

Applications of Recommendation Systems

- Product recommendations
  - Amazon or similar online vendors
- Movie recommendations
  - Netflix offers its customers recommendations of movies they might like
- News articles
  - News services have attempted to identify articles of interest to readers based on the articles that they have read in the past
  - Blogs, YouTube

Netflix Prize

- The Netflix Prize challenge concerned recommender systems for movies (October, 2006)
- Netflix released a training set consisting of data from almost 500,000 customers and their ratings on 18,000 movies.
  - More than 100 million ratings
- The task was to use these data to build a model to predict ratings for a hold-out set of 3 million ratings

Large scale data analysis using Spark
CASE STUDY: Recommending Music and the Audioscrobbler Dataset
Dataset

Dataset

- Audioscrobbler dataset
  - 2002, Richard Jones
  - Collecting and analyzing user's songs to generate recommendation
  - Started with support for Winamp and XMMS
  - iTunes, Winamp, Windows Media Player, Foobar, iPod, Amarok, Rhythmbox, mpd, Xbox media center, Slimserver, Jinzora, mpg321, Muine, Rhapsody, YME, Soundbridge, VLC…

last.fm

Dataset

- Confined rating system
  - “Bob **rates** Coldplay 3.5 stars.”
  - Users rate music far less frequently than they play music
- Audioscrobbler dataset
  - “Bob **played** Coldplay track”
  - Each individual data carries less information
- Implicit feedback
  - User-artist connections are implied as a side effect of other actions

Dataset

- 141,000 unique users
- 1.6 million unique artists
- 24.2 million user’s plays of artist are recorded
  - User_artist_data.txt
  - http://www-etud.iro.umontreal.ca/~bergstrj/audioscrobbler_data.html
- On average, each user has played songs from about 171 artists (out of 1.6 M)
  - Extremely sparse dataset

Large scale data analysis using Spark
CASE STUDY: Recommending Music and the Audioscrobbler Dataset
Collaborative Filtering

Collaborative filtering [1/2]

- Collects and analyzes a large amount of information on users’ behaviors, activities or preferences and predicts what users will like based on their similarity to other users
- Explicit data collection
  - Rate an item
  - Search history
  - Favorite item
  - Wish list
- Implicit data collection
  - Viewing times
  - Tracking online purchases
  - Analyzing the user’s social network

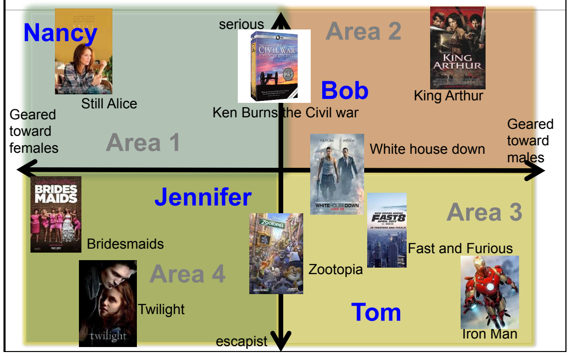
Collaborative filtering [2/2]

- Two users may share similar tastes because they are the same age
  - It is **NOT** an example of collaborative filtering
- Two users may both like the same song because they play many other same songs
  - It **IS** an example of collaborative filtering
- Algorithm that learns <u>without</u> access to user or artist attributes

Latent-Factor model (1/2)

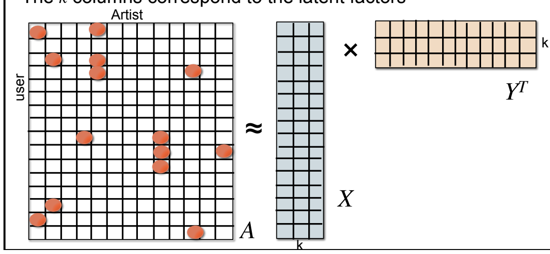
- Tries to explain **observed interactions** between large numbers of users and products through a relatively small number of **unobserved, underlying reasons**
- Within the music business context,
  - Why **millions of people** buy a particular **few of thousands of possible albums** by describing users and albums for **tens of genres and tastes that are not directly observable**

Simplified illustration of the latent factor approach



How do we model this?

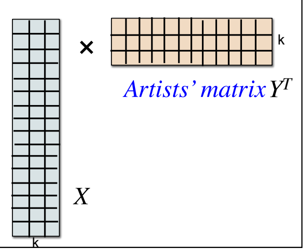
- User and product data in a large matrix A
- Row i and column j
  - If user i has played artist j
- The k columns correspond to the latent factors



Creating user and artist matrices

- Two matrices
  - Matrix X for user
    - Each value corresponds to a latent feature in the model
  - Matrix Y for artist
    - Each value corresponds to a latent feature in the model
  - Rows express how much users and artists associate with these latent features
- Product of X and Y
  - Complete estimation of the entire, dense user-artist interaction matrix

Users' matrix A



Computational challenge

- $A=XY^T$ generally no solution
  - X and Y are not large enough
- Goal
  - **Finding the best X and Y**

Alternating Least Squares (ALS)

- Alternating least squares algorithm to compute X and Y
  - Spark Mlib's ALS implementation
- Step 1
  - Y is not known
  - Initialized to a matrix with randomly chosen row vectors
  - Then simple linear algebra gives the best X, given Y and A
  - $A_iY(Y^TY)^{-1}=X^i$
  - Equality cannot achieved exactly
    - The goal becomes to minimize $|A_iY(Y^TY)^{-1} - X^i|$
    - The sum of squared differences between the two matrices' entries

Alternating Least Squares (ALS)

- Step 2.
  - Repeat similar sequence as step 1 to compute Y from the X (from step 1)
- Step 3.
  - Repeat similar sequence as step 1 to compute X from the Y (from step 2)

. . .

- X and Y do eventually **converge** to good (acceptable) solutions

Alternating Least Squares (ALS)

- Takes advantage of the sparsity of the input data
- Easy to apply data parallelism

Large scale data analysis using Spark

CASE STUDY: Recommending Music and the Audioscrobbler Dataset

Building a model with Spark MLib

Preparing the Data

- Files are available at /user/ds/
- Spark MLib's ALS implementation
  - Requires numeric IDs for users and items
  - Nonnegative 32-bit integers
  - An ID larger than Integer.MAX_VALUE **cannot** be used

```
val rawUserArtistData =
      sc.textFile("hdfs:///user/ds/user_artist_data.txt")

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

Maximum user IDs: 24443548
Maximum artist IDs: 2147483647
No additional transformation will be needed
```


Extracting names

- `artist_data.txt`
- Artist ID and name separated by a tab

```
val rawArtistData =
 sc.textFile(" hdfs:///user/ds/artist_data.txt")

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

- Straightforward parsing of the file into `(Int, String)` tuples will fail

Extracting names

- Scala's `Option` class
  - `Option` represents a value that might only optionally exist

```
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
       }
    }
}
```


Building a First Model

- Two transformations are required
  - Aliases dataset should be applied to convert all artist IDs to a canonical ID
  - The data should be converted to a Rating object
    - User-product-value data

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

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

cache()

- RDD should be temporarily stored after being computed
- ALS is iterative
  - It will typically need to access this RDD ≥ 10 times
  - Otherwise, this RDD could be repeatedly recomputed from the original data each time

Storage Level	Cached Partitions	Fraction Cached	Size in Memory
Memory Deserialized 1x Replicated	120	100%	886.8 MB

Broadcast variables

- For the case that many tasks (from different closures) need access to the same (immutable) data structure
- Extends normal handling of task closures
  - Caching data as raw Java objects on each executor
  - Caching data across multiple jobs and stages
- Spark will send, and hold in memory, just one copy for each executor in the cluster
  - Saves network traffic and memory

Building the ALS model

- Constructs `model` as a `MatrixFactorizationModel`

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


Retrieving some feature vectors

- Array of 10 numbers

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

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

 ...
(4293,-0.3233030601963864, 0.319645275935413 25,
0.49025505511361034, 0.09000932568001832, 0.4429537767744912,
0.4186675713407441, 0.8026858843673894, -0.4841300444834003,
-0.12485901532338621, 0.19795451025931002)
```


Spot Checking Recommendations

- To see if the artist recommendations for user(2093760) makes any intuitive sense

```
val rawArtistsForUser =
  rawUserArtistData.map(_. split(' ')).
   filter { case Array( user,_,_) = > user.toInt = = 2093760 }

val existingProducts = rawArtistsForUser.map { case
   Array(_, artist,_) = > artist.toInt }.collect().toSet

artistByID.filter { case (id, name) = >
   existingProducts.contains(id)
}.values.collect().Foreach(println)
...
David Gray
Blackalicious
Jurassic
The Saw Doctors
Xzibit
```


Spot Checking Recommendations

- To see five recommendations for this user (ID: `2093760`)

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


...

Rating( 2093760,1300642,0.02833118412903932)
Rating( 2093760,2814,0.027832682960168387)
Rating( 2093760,1037970,0.02726611004625264)
Rating( 2093760,1001819,0.02716011293509426)
Rating( 2093760,4605,0.027118271894797333)
```

Large scale data analysis using Spark
CASE STUDY: Recommending Music and the Audioscrobbler Dataset
Evaluating the Recommendation Model

What is a "**good**" recommendation?

- "a popular artist"?
- "artists the user has listened to"?
- "artists the user will listen to"?

Preparing data for evaluation

- To perform a meaningful evaluation, some of the artist play data can be set aside
  - Hidden from the ALS model building process
- The held-out data can be used as a collection of good recommendations for each user
- Compute the recommender's score

For building model	For testing model