APR 1ST, 2013

Movie Recommendations and More With Spark

This post is inspired by Edwin Chen's post on Scalding. I encourage you to first read that post! The Spark code is adapted from his Scalding code and is available in full here.

As outlined in Ed's post, Scalding is a Scala DSL for Hadoop MapReduce that makes it easier, more natural and more concise to write MapReduce workflows. The Scala code ultimately compiles down to MapReduce jobs via Cascading.

Spark

The Spark Project is a cluster computing framework that emphasizes low-latency job execution and in-memory caching to provide speed. It can run up to 100x faster than Hadoop MapReduce (when all the data is cached in memory) as a result. It is written in Scala, but also has Java and Python APIs. It is fully compatible with HDFS and any Hadoop `InputFormat/OutputFormat`, but is independent of Hadoop MapReduce.

The Spark API bears many similarities to Scalding, providing a way to write natural Scala code instead of `Mappers` and `Reducers`. Taking Ed's example:

Scalding

```scala
// Create a histogram of tweet lengths.
tweets.map('tweet -> 'length) { tweet : String => tweet.size }.groupBy('length) { _.size }
```

Spark

```scala
// Create a histogram of tweet lengths.
tweets.groupBy(tweet : String => tweet.size).map(pair => (pair._1, pair._2.size))
```

Movie Similarities

I've recently been experimenting a lot with Spark, and thought it would be interesting to compare Ed's approach to computing movie similarities in Scalding with Spark. So I've ported his Scalding code over to Spark and we'll compare the two as we go along. For a basic introduction to Spark's API see the Spark Quickstart.

Firstly, we read the ratings from a file. Since I don't have access to a nice Twitter tweet datasource, I used the MovieLens 100k rating dataset. The training set ratings are in a file called `ua.base`, while the movie item data is in `u.item`.

Scalding

```scala
/**
 * The input is a TSV file with three columns: (user, movie, rating).
 */
val INPUT_FILENAME = "data/ratings.tsv"

/**
 * Read in the input and give each field a type and name.
 */
val ratings = Tsv(INPUT_FILENAME, ('user, 'movie, 'rating))

/**
 * Let's also keep track of the total number of people who rated each movie.
 */
val numRaters =
  ratings
    // Put the number of people who rated each movie into a field called "numRaters".
    .groupBy('movie) { _.size }.rename('size -> 'numRaters)

// Merge `ratings` with `numRaters`, by joining on their movie fields.
val ratingsWithSize =
  ratings.joinWithSmaller('movie -> 'movie, numRaters)

// ratingsWithSize now contains the following fields: (user, movie, rating, numRaters).
```

Spark

```scala
val TRAIN_FILENAME = "ua.base"
val MOVIES_FILENAME = "u.item"

// Spark programs require a SparkContext to be initialized
val sc = new SparkContext(master, "MovieSimilarities")

// extract (userid, movieid, rating) from ratings data
val ratings = sc.textFile(TRAIN_FILENAME)
  .map(line => {
    val fields = line.split("\t")
    (fields(0).toInt, fields(1).toInt, fields(2).toInt)
})

// get num raters per movie, keyed on movie id
val numRatersPerMovie = ratings
  .groupBy(tup => tup._2)
  .map(grouped => (grouped._1, grouped._2.size))

// join ratings with num raters on movie id
val ratingsWithSize = ratings
  .groupBy(tup => tup._2)
  .join(numRatersPerMovie)
  .flatMap(joined => {
    joined._2._1.map(f => (f._1, f._2, f._3, joined._2._2))
})

// ratingsWithSize now contains the following fields: (user, movie, rating, numRaters).
```

Similarly to Scalding's `Tsv` method, which reads a TSV file from HDFS, Spark's `sc.textFile` method reads a text file from HDFS. However it's up to us to specify how to split the fields.

Also, Spark's API for joins is a little lower-level than Scalding's, hence we have to `groupBy` first and transform after the `join` with a `flatMap` operation to get the fields we want. Scalding actually does something similar under the hood of `joinWithSmaller`.

Computing similarity

In order to determine how similar two movies are to each other, we must (as per Ed's post again):

> - *For every pair of movies A and B, find all the people who rated both A and B.*
> - *Use these ratings to form a Movie A vector and a Movie B vector.*
> - *Calculate the correlation between these two vectors.*
> - *Whenever someone watches a movie, you can then recommend the movies most correlated with it.*

This is item-based collaborative filtering. So let's compute the first two steps above:

Scalding

```scala
/**
 * To get all pairs of co-rated movies, we'll join `ratings` against itself.
 * So first make a dummy copy of the ratings that we can join against.
 */
val ratings2 =
  ratingsWithSize
    .rename(('user, 'movie, 'rating, 'numRaters) -> ('user2, 'movie2, 'rating2, 'numRaters2))

/**
 * Now find all pairs of co-rated movies (pairs of movies that a user has rated) by
 * joining the duplicate rating streams on their user fields,
 */
val ratingPairs =
  ratingsWithSize
    .joinWithSmaller('user -> 'user2, ratings2)
    // De-dupe so that we don't calculate similarity of both (A, B) and (B, A).
    .filter('movie, 'movie2) { movies : (String, String) => movies._1 < movies._2 }
    .project('movie, 'rating, 'numRaters, 'movie2, 'rating2, 'numRaters2)

// By grouping on ('movie, 'movie2), we can now get all the people who rated any pair of movies
```

Spark

```scala
// dummy copy of ratings for self join
val ratings2 = ratingsWithSize.keyBy(tup => tup._1)

// join on userid and filter movie pairs such that we don't double-count and exclude self-pairs
val ratingPairs =
  ratingsWithSize
  .keyBy(tup => tup._1)
  .join(ratings2)
  .filter(f => f._2._1._2 < f._2._2._2)
```

Notice how similar the APIs are with respect to the functional operations like `filter` - they each simply take a Scala closure. We then compute the various vector metrics for each ratings vector (size, dot-product, norm etc). We'll use these to compute the various similarity metrics between pairs of movies.

Scalding

```scala
/**
 * Compute dot products, norms, sums, and sizes of the rating vectors.
 */
val vectorCalcs =
  ratingPairs
    // Compute (x*y, x^2, y^2), which we need for dot products and norms.
    .map(('rating, 'rating2) -> ('ratingProd, 'ratingSq, 'rating2Sq)) {
      ratings : (Double, Double) =>
      (ratings._1 * ratings._2, math.pow(ratings._1, 2), math.pow(ratings._2, 2))
    }
    .groupBy('movie, 'movie2) { group =>
        group.size // length of each vector
        .sum('ratingProd -> 'dotProduct)
        .sum('rating -> 'ratingSum)
        .sum('rating2 -> 'rating2Sum)
        .sum('ratingSq -> 'ratingNormSq)
        .sum('rating2Sq -> 'rating2NormSq)
        .max('numRaters) // Just an easy way to make sure the numRaters field stays.
        .max('numRaters2)
        // All of these operations chain together like in a builder object.
    }
```

Spark

```scala
// compute raw inputs to similarity metrics for each movie pair
val vectorCalcs =
  ratingPairs
  .map(data => {
    val key = (data._2._1._2, data._2._2._2)
    val stats =
      (data._2._1._3 * data._2._2._3, // rating 1 * rating 2
        data._2._1._3,                // rating movie 1
        data._2._2._3,                // rating movie 2
        math.pow(data._2._1._3, 2),   // square of rating movie 1
        math.pow(data._2._2._3, 2),   // square of rating movie 2
        data._2._1._4,                // number of raters movie 1
        data._2._2._4)                // number of raters movie 2
    (key, stats)
  })
  .groupByKey()
  .map(data => {
    val key = data._1
    val vals = data._2
    val size = vals.size
    val dotProduct = vals.map(f => f._1).sum
    val ratingSum = vals.map(f => f._2).sum
    val rating2Sum = vals.map(f => f._3).sum
    val ratingSq = vals.map(f => f._4).sum
    val rating2Sq = vals.map(f => f._5).sum
    val numRaters = vals.map(f => f._6).max
    val numRaters2 = vals.map(f => f._7).max
    (key, (size, dotProduct, ratingSum, rating2Sum, ratingSq, rating2Sq, numRaters, numRaters2))
  })
```

Similarity metrics

For each movie pair we compute *correlation*, *regularized correlation*, *cosine similarity* and *Jaccard similarity* (see Ed's post and the code for full details).

Scalding

```scala
val PRIOR_COUNT = 10
val PRIOR_CORRELATION = 0

val similarities =
  vectorCalcs
    .map(('size, 'dotProduct, 'ratingSum, 'rating2Sum, 'ratingNormSq, 'rating2NormSq, 'numRater
      ('correlation, 'regularizedCorrelation, 'cosineSimilarity, 'jaccardSimilarity)) {

      fields : (Double, Double, Double, Double, Double, Double, Double, Double) =>

      val (size, dotProduct, ratingSum, rating2Sum, ratingNormSq, rating2NormSq, numRaters, nu

      val corr = correlation(size, dotProduct, ratingSum, rating2Sum, ratingNormSq, rating2Norm
      val regCorr = regularizedCorrelation(size, dotProduct, ratingSum, rating2Sum, ratingNorm
      val cosSim = cosineSimilarity(dotProduct, math.sqrt(ratingNormSq), math.sqrt(rating2Norm
      val jaccard = jaccardSimilarity(size, numRaters, numRaters2)

      (corr, regCorr, cosSim, jaccard)
    }
```

Spark

```scala
val PRIOR_COUNT = 10
val PRIOR_CORRELATION = 0

// compute similarity metrics for each movie pair
val similarities =
  vectorCalcs
  .map(fields => {

    val key = fields._1
    val (size, dotProduct, ratingSum, rating2Sum, ratingNormSq, rating2NormSq, numRaters, numRa

    val corr = correlation(size, dotProduct, ratingSum, rating2Sum, ratingNormSq, rating2NormSq
    val regCorr = regularizedCorrelation(size, dotProduct, ratingSum, rating2Sum,
      ratingNormSq, rating2NormSq, PRIOR_COUNT, PRIOR_CORRELATION)
    val cosSim = cosineSimilarity(dotProduct, scala.math.sqrt(ratingNormSq), scala.math.sqrt(ra
    val jaccard = jaccardSimilarity(size, numRaters, numRaters2)

    (key, (corr, regCorr, cosSim, jaccard))
  })
```

The nice thing here is that, once the raw input metrics themselves are computed, we can use the exact same functions from the Scalding example to compute the similarity metrics as can be seen above - I simply copy-and-pasted Ed's `correlation`, `regularizedCorrelation`, `cosineSimilarity` and `jaccardSimilarity` functions!

Some results

So, what do the results look like after putting all of this together? Since I used a different input data source we won't get the same results, but we'd hope that most of them would make sense. Similarly to Ed's results, I found that using raw `correlation` resulted in sub-optimal similarities (at least from eye-balling and "sense checking"), since some movie pairs have very few common raters (many had just 1 rater in common).

I also found that `cosine similarity` didn't do so well on a "sense check" basis either, which was somewhat surprising since this is usually the standard similarity metric for collaborative filtering. This seems to be due to a lot of movies having cosine similarity of 1.0, so perhaps I have messed up the calculation somewhere (if you spot an error please let me know).

In any case, here are the top 10 movies most similar to Die Hard (1998), ranked by `regularized correlation`:

Movie 1	Movie 2	Correlation	Reg Correlation	Cosine Similarity	Jaccard Similarity
Die Hard (1988)	Die Hard: With a Vengeance (1995)	0.5413	0.4946	0.9692	0.4015
Die Hard (1988)	Die Hard 2 (1990)	0.4868	0.4469	0.9687	0.4088
Die Hard (1988)	Bananas (1971)	0.5516	0.4390	0.9745	0.1618
Die Hard (1988)	Good, The Bad and The Ugly, The (1966)	0.4608	0.4032	0.9743	0.2518
Die Hard (1988)	Hunt for Red October, The (1990)	0.4260	0.3944	0.9721	0.4098
Die Hard (1988)	City Slickers II: The Legend of Curly's Gold (1994)	0.5349	0.3903	0.9506	0.1116
Die Hard (1988)	Grease 2 (1982)	0.6502	0.3901	0.9449	0.0647
Die Hard (1988)	Star Trek: The Wrath of Khan (1982)	0.4160	0.3881	0.9675	0.4441
Die Hard (1988)	Sphere (1998)	0.7722	0.3861	0.9893	0.0403
Die Hard (1988)	Field of Dreams (1989)	0.4126	0.3774	0.9630	0.3375

Looks fairly reasonable! And here are the 10 most similar to Il Postino:

Movie 1	Movie 2	Correlation	Reg Correlation	Cosine Similarity	Jaccard Similarity
Postino, Il (1994)	Bottle Rocket (1996)	0.8789	0.4967	0.9855	0.0699
Postino, Il (1994)	Looking for Richard (1996)	0.7112	0.4818	0.9820	0.1123
Postino, Il (1994)	Ridicule (1996)	0.6550	0.4780	0.9759	0.1561
Postino, Il (1994)	When We Were Kings (1996)	0.7581	0.4773	0.9888	0.0929
Postino, Il (1994)	Mother Night (1996)	0.8802	0.4611	0.9848	0.0643
Postino, Il (1994)	Kiss Me, Guido (1997)	0.9759	0.4337	0.9974	0.0452
Postino, Il (1994)	Blue in the Face (1995)	0.6372	0.4317	0.9585	0.1148
Postino, Il (1994)	Othello (1995)	0.5875	0.4287	0.9774	0.1330
Postino, Il (1994)	English Patient, The (1996)	0.4586	0.4210	0.9603	0.2494
Postino, Il (1994)	Mediterraneo (1991)	0.6200	0.4200	0.9879	0.1235

How about Star Wars?

Movie 1	Movie 2	Correlation	Reg Correlation	Cosine Similarity	Jaccard Similarity
Star Wars (1977)	Empire Strikes Back, The (1980)	0.7419	0.7168	0.9888	0.5306
Star Wars (1977)	Return of the Jedi (1983)	0.6714	0.6539	0.9851	0.6708
Star Wars (1977)	Raiders of the Lost Ark (1981)	0.5074	0.4917	0.9816	0.5607
Star Wars (1977)	Meet John Doe (1941)	0.6396	0.4397	0.9840	0.0442
Star Wars (1977)	Love in the Afternoon (1957)	0.9234	0.4374	0.9912	0.0181
Star Wars (1977)	Man of the Year (1995)	1.0000	0.4118	0.9995	0.0141
Star Wars (1977)	When We Were Kings (1996)	0.5278	0.4021	0.9737	0.0637
Star Wars (1977)	Cry, the Beloved Country (1995)	0.7001	0.3957	0.9763	0.0257
Star Wars (1977)	To Be or Not to Be (1942)	0.6999	0.3956	0.9847	0.0261
Star Wars (1977)	Haunted World of Edward D. Wood Jr., The (1995)	0.6891	0.3895	0.9758	0.0262

Finally, what about the 10 most *dissimilar* to Star Wars?

Movie 1	Movie 2	Correlation	Reg Correlation	Cosine Similarity	Jaccard Similarity
Star Wars (1977)	Fathers' Day (1997)	-0.6625	-0.4417	0.9074	0.0397
Star Wars (1977)	Jason's Lyric (1994)	-0.9661	-0.3978	0.8110	0.0141
Star Wars (1977)	Lightning Jack (1994)	-0.7906	-0.3953	0.9361	0.0202
Star Wars (1977)	Marked for Death (1990)	-0.5922	-0.3807	0.8729	0.0361
Star Wars (1977)	Mixed Nuts (1994)	-0.6219	-0.3731	0.8806	0.0303
Star Wars (1977)	Poison Ivy II (1995)	-0.7443	-0.3722	0.7169	0.0201
Star Wars (1977)	In the Realm of the Senses (Ai no corrida) (1976)	-0.8090	-0.3596	0.8108	0.0162
Star Wars (1977)	What Happened Was... (1994)	-0.9045	-0.3392	0.8781	0.0121
Star Wars (1977)	Female Perversions (1996)	-0.8039	-0.3310	0.8670	0.0141
Star Wars (1977)	Celtic Pride (1996)	-0.6062	-0.3175	0.8998	0.0220

I'll leave it to you to decide on the accuracy.

Conclusion and Next Steps

Hopefully this gives a taste for Spark and how it can be used in a very similar manner to Scalding and MapReduce - with all the advantages of HDFS compatability, in-memory caching capabilities, low-latency execution and other distributed-memory primitives (such as broadcast variables and accumulators); not to mention interactive analysis via the Scala/Spark console, and a Java and Python API! Check out the documentation, tutorials and examples here.

One issue that is apparent from the above code snippets is that Scalding's API is somewhat cleaner when doing complex field manipulations and joins, due to the ability to have named fields as Scala `Symbols`, e.g. `tweets.map('tweet -> 'length) { tweet : String => tweet.size }`

The lack of named fields in Spark's API does lead to some messy tuple-unpacking and makes keeping track of which fields are which more complex. This could be an interesting potential addition to Spark.

Finally, please do let me know if you find any issues or errors. And thanks to the Spark team for a fantastic project!

Posted by Nick Pentreath • Apr 1st, 2013 • Scala, Spark, collaborative filtering, recommendations

Tweet

Recent Posts

Movie recommendations and more with Spark

GitHub Repos

elasticsearch-vector-scoring
Score documents with pure dot product / cosine similarity with ES

beijing-meetup-2016
Jupyter Notebooks for Apache Spark ML Meetup - Beijing Nov 2016

glint-fm
Factorization Machines on Spark and Glint

mlnick.github.com

hive-udf
Approximate cardinality estimation with HyperLogLog, as a Hive function

@MLnick on GitHub

Latest Tweets

Status updating...

Follow @MLnick