Master Executive di II Livello

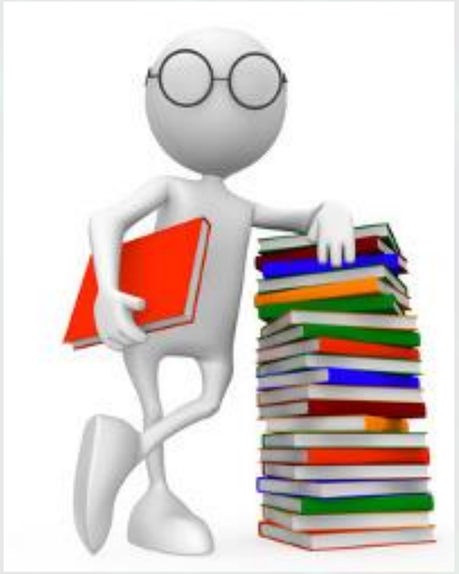
BIG DATA ANALYSIS AND BUSINESS INTELLIGENCE

Vamsi Krishna Varma Gunturi
Data science intern at ISTAT
vamsivarmagunturi@gmail.com

Spark in action

fondazione INUIT TORVERGATA

Topics



- Resilient distributed dataset (RDD)
- Creating RDD’s
- Transforming RDD’s
- RDD actions
- Lazy evaluation
- **Demo**: Lowest average ratings using RDDs
- Spark SQL
- **Demo**: Lowest average ratings using Spark SQL
- Spark Mllib
- **Demo**: Movie recommendations using MLlib

RDD

- **R**esilient **D**istributed **D**ataset
- Read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way.
- It's an abstraction across all the process that happens under the hood in Spark.
- To make sure your job is evenly distributed across your cluster that it can handle failures in a resilient manner.
- And at the end of the day it just looks like a data set to you.

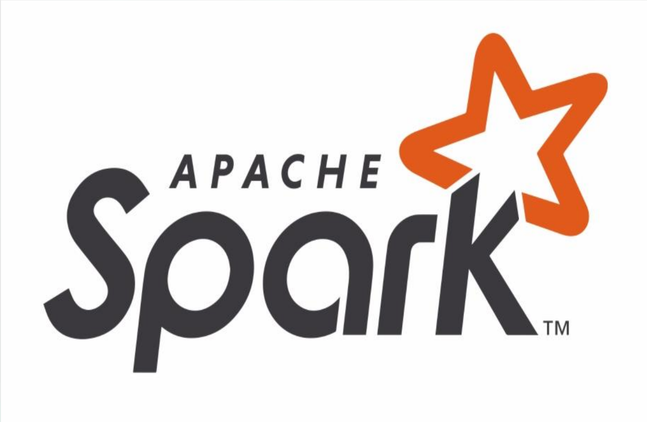
SparkContext

- Created by your driver program.
- Responsible for making RDD's resilient and distributed. Basically an environment that you run RDD's in.
- Creates RDD's
- The Spark shell creates a "sc" object for you

Creating RDD's

- nums = parallelize([1,2,3,4])
- sc.textFile(file://c:/users/francesco/big-text-file.txt) or s3n://, hdfs://
- hiveCtx = HiveContext(sc) rows = hiveCtx.sql("SELECT name, age FROM users")
- Can also create from:
  - JDBC
  - Cassandra
  - Hbase
  - Elasticsearch
  - JSON, CSV, sequence files, object files, various compressed formats

Transforming RDD's



- **map** – one to one relationship, maps every row in the RDD and transforms it to another RDD
- **flatmap** – any relationship, Ex: map an input row in to multiple lines
- **filter**
- **distinct**
- **sample** – sample RDD's randomly
- **union**, **intersection**, **subtract**, **cartesian** – Combine RDD's

Map example

- Example: Taking an RDD of integers and squaring them

```
rdd = sc.parallelize([1,2,3,4])
squaredRDD = rdd.map(lambda x: x*x)
```

This yields 1,4,9,16

- Many RDD methods accept a function as a parameter

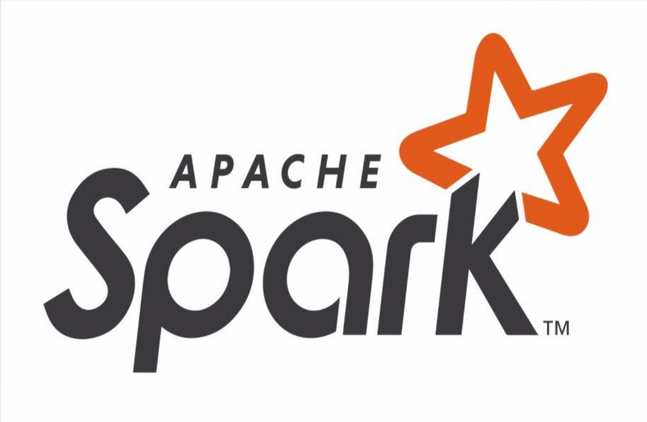
```
rdd.map(lambda x: x*x)
```

Is the same thing as,

```
def squareIt(x):
    return x*x

rdd.map(squareIt)
```

RDD actions



- RDD actions are for reducing the mapped data.
- Below are some functions -

collect – For outputting the data in RDD to driver program

count – Count for number of rows in RDD

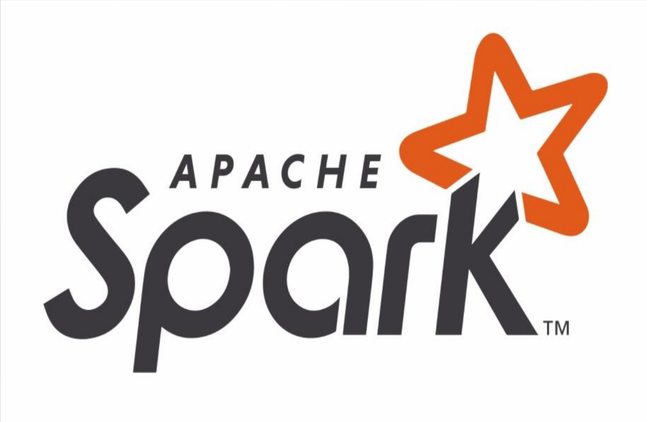
countByValue – Tally by each unique value in RDD (similar to GroupBy)

take – For debugging, return a portion of data (sample of entire data)

top

reduce – Helps you write a function to combine all the values associated with a unique key

Lazy Evaluation



- In spark nothing actually happens in your driver program until one of these actions are called. This is referred to as Lazy evaluation.
- One you invoke an action, you are basically telling Spark that this is the result I am expecting and then Spark will work backwards and figure out the fastest way to achieve the result that we are expecting.
- So basically until you hit an action, all Spark is doing is building a chain of dependencies with in your driver script
- Only when action is invoked, Spark will figure out the quickest path through those dependencies and actually at this point that it kicks off the job on your cluster.
- As you can see, this workflow is confusing when you are debugging but this is one of the secrets of Spark speed.

Example: Find worst movies from movielens dataset

LowestRatedMovieSpark.py

```python
from pyspark import SparkConf, SparkContext

# This function just creates a Python "dictionary" we can later
# use to convert movie ID's to movie names while printing out
# the final results.
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1]
    return movieNames

# Take each line of u.data and convert it to (movieID, (rating, 1.0))
# This way we can then add up all the ratings for each movie, and
# the total number of ratings for each movie (which lets us compute the average)
def parseInput(line):
    fields = line.split()
    return (int(fields[1]), (float(fields[2]), 1.0))

if __name__ == "__main__":
    # The main script - create our SparkContext
    conf = SparkConf().setAppName("WorstMovies")
    sc = SparkContext(conf = conf)

    # Load up our movie ID -> movie name lookup table
    movieNames = loadMovieNames()
```


Vamsi Krishna Varma Gunturi

Example: Find worst movies from movielens dataset

```
24      sc = SparkContext(conf = conf)
25
26      # Load up our movie ID -> movie name lookup table
27      movieNames = loadMovieNames()
28
29      # Load up the raw u.data file
30      lines = sc.textFile("hdfs:///user/maria_dev/ml-100k/u.data")
31
32      # Convert to (movieID, (rating, 1.0))
33      movieRatings = lines.map(parseInput)
34
35      # Reduce to (movieID, (sumOfRatings, totalRatings))
36      ratingTotalsAndCount = movieRatings.reduceByKey(lambda movie1, movie2: ( movie1[0] + movie2[
37
38      # Map to (movieID, averageRating)
39      averageRatings = ratingTotalsAndCount.mapValues(lambda totalAndCount : totalAndCount[0] / to
40
41      # Sort by average rating
42      sortedMovies = averageRatings.sortBy(lambda x: x[1])
43
44      # Take the top 10 results
45      results = sortedMovies.take(10)
46
47      # Print them out:
48      for result in results:
49          print(movieNames[result[0]], result[1])
50
```


Vamsi Krishna Varma Gunturi

Spark SQL

- Makes use of DataFrames and Datasets.
- Mainly for working with structured data.
- Extends RDD to a “DataFrame” object.
- DataFrames:
  - Contains Row objects
  - Can run SQL queries
  - Has a schema (leading to more efficient storage)
  - Read and write to JSON, Hive, parquet
  - Communicates with JDBC/ODBC, Tableau.

Spark SQL with Python

- from pyspark.sql import SQLContext, Row
- hiveContext = HiveContext(sc)
- inputData = spark.read.json(datafile)
- inputData.createOrReplaceTempView(“structed_data”)
- resultDataFrame = hiveContext.sql(“”””SELECT name FROM EMPLOYEES ORDER BY salary”””””)

Spark SQL with Python

Other stuff you can do with dataframes

- resultDataFrame.show()
- resultDataFrame.select("field_name)
- resultDataFrame.filter(resultDataFram("field_name" > 200))
- resultDataFrame.groupBy(resultDataFrame("field_name")).mean()
- resultDataFrame.rdd().map(mapperFunction)

Example: Find worst movies from movielens dataset

LowestRatedMovieDataFrame.py

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql import Row
3 from pyspark.sql import functions
4
5 def loadMovieNames():
6     movieNames = {}
7     with open("ml-100k/u.item") as f:
8         for line in f:
9             fields = line.split('|')
10            movieNames[int(fields[0])] = fields[1]
11    return movieNames
12
13 def parseInput(line):
14    fields = line.split()
15    return Row(movieID = int(fields[1]), rating = float(fields[2]))
16
17 if __name__ == "__main__":
18    # Create a SparkSession
19    spark = SparkSession.builder.appName("PopularMovies").getOrCreate()
20
21    # Load up our movie ID -> name dictionary
22    movieNames = loadMovieNames()
23
24    # Get the raw data
25    lines = spark.sparkContext.textFile("hdfs:///user/maria_dev/ml-100k/u.data")
26    # Convert it to a RDD of Row objects with (movieID, rating)
27    movies = lines.map(parseInput)
```

Example: Find worst movies from movielens dataset

```
movieNames = loadMovieNames()

# Get the raw data
lines = spark.sparkContext.textFile("hdfs:///user/maria_dev/ml-100k/u.data")
# Convert it to a RDD of Row objects with (movieID, rating)
movies = lines.map(parseInput)
# Convert that to a DataFrame
movieDataset = spark.createDataFrame(movies)

# Compute average rating for each movieID
averageRatings = movieDataset.groupBy("movieID").avg("rating")

# Compute count of ratings for each movieID
counts = movieDataset.groupBy("movieID").count()

# Join the two together (We now have movieID, avg(rating), and count columns)
averagesAndCounts = counts.join(averageRatings, "movieID")

# Pull the top 10 results
topTen = averagesAndCounts.orderBy("avg(rating)").take(10)

# Print them out, converting movie ID's to names as we go.
for movie in topTen:
    print (movieNames[movie[0]], movie[1], movie[2])

# Stop the session
spark.stop()
```

Spark MLlib

- Machine learning library for Spark
- Capabilities –

- Feature extraction:
  Term frequency / Inverse document frequency useful for search
- Basic statistics
- Principal component analysis, singular value decomposition
- Recommendations using Alternating least squares
- Decision trees
- K-means clustering
- Regression, Support vector machines

MLlib Example: Movie recommendations

```
from pyspark.sql import SparkSession
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row
from pyspark.sql.functions import lit

# Load up movie ID -> movie name dictionary
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.item") as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1].decode('ascii', 'ignore')
    return movieNames

# Convert u.data lines into (userID, movieID, rating) rows
def parseInput(line):
    fields = line.value.split()
    return Row(userID = int(fields[0]), movieID = int(fields[1]), rating = float(fields[2]))


if __name__ == "__main__":
    # Create a SparkSession (the config bit is only for Windows!)
    spark = SparkSession.builder.appName("MovieRecs").getOrCreate()

    # Load up our movie ID -> name dictionary
    movieNames = loadMovieNames()
```


Vamsi Krishna Varma Gunturi

MLlib Example: Movie recommendations

```
# Load up our movie ID -> name dictionary
movieNames = loadMovieNames()

# Get the raw data
lines = spark.read.text("hdfs:///user/maria_dev/ml-100k/u.data").rdd

# Convert it to a RDD of Row objects with (userID, movieID, rating)
ratingsRDD = lines.map(parseInput)

# Convert to a DataFrame and cache it
ratings = spark.createDataFrame(ratingsRDD).cache()

# Create an ALS collaborative filtering model from the complete data set
als = ALS(maxIter=5, regParam=0.01, userCol="userID", itemCol="movieID", ratingCol="rating")
model = als.fit(ratings)

# Print out ratings from user 0:
print("\nRatings for user ID 0:")
userRatings = ratings.filter("userID = 0")
for rating in userRatings.collect():
    print movieNames[rating['movieID']], rating['rating']

print("\nTop 20 recommendations:")
# Find movies rated more than 100 times
ratingCounts = ratings.groupBy("movieID").count().filter("count > 100")
# Construct a "test" dataframe for user 0 with every movie rated more than 100 times
popularMovies = ratingCounts.select("movieID").withColumn('userID', lit(0))
```

MLlib Example: Movie recommendations

```
print("\nTop 20 recommendations:")
# Find movies rated more than 100 times
ratingCounts = ratings.groupBy("movieID").count().filter("count > 100")
# Construct a "test" dataframe for user 0 with every movie rated more than 100 times
popularMovies = ratingCounts.select("movieID").withColumn('userID', lit(0))

# Run our model on that list of popular movies for user ID 0
recommendations = model.transform(popularMovies)

# Get the top 20 movies with the highest predicted rating for this user
topRecommendations = recommendations.sort(recommendations.prediction.desc()).take(20)

for recommendation in topRecommendations:
    print (movieNames[recommendation['movieID']], recommendation['prediction'])

spark.stop()
```

Master Executive di II Livello

BIG DATA ANALYSIS AND BUSINESS INTELLIGENCE

Vamsi Krishna Varma Gunturi
Data science intern at ISTAT
vamsivarmagunturi@gmail.com

Grazie

fondazione INUIT TORVERGATA