Big Data Warehouses and Analytics: About Scalability and Realtime

Pedro Furtado  
Sixth European Business Intelligence & Big Data Summer School (eBISS 2016)

Roadmap
1. Generic Data Organizations and Architectures
2. Spark
3. Lambda Architecture and Realtime
4. Analytics and Data Mining with Spark

What types of apps are there? NOSQL? FRAMEWORKS?
Big Data Stores
- Huge data Scale
  Latency < 1 sec

Transactional/operational systems
- Always Available
  ACID
  Latency < 100 msecs

Analytic/Decision support systems
- Scale complex relational apps
  Latency < 5 sec
  Elastic

Data mining
- Procedural flexibility
  Scalable, elastic

Relational DBs
- Page
  Row:
  RowID: n RowO:
  col-wise:
  index values
  1: col-value

1. RDBMS and Beyond: Optimization
RDBMS Query Plans and Optimization
Whatever you may have (key-val, doc, hadoop),
can you match RDBMS optimization capabilities?

Explain Select B,D From R join S on C
Where R.A = 'c' and S.E = 2
and R.C=S.C

Presentation
Pedro Furtado, U. Coimbra, Portugal
Critique of Relational DBs

Fallacy: That it is mostly disk-based

Most Severe Limitation: Lots of housekeeping overheads:

RDBMS processing profile (time spent)

12% useful work

20% logging
18% locking
10% buffer management
29% index management

OlTP through the Looking Glass, and what we found there, S. Harizopoulos, D. Abadi, S. Madden and M. Stonebraker, in ACM SIGMOD 2008

Key-value Stores

Most basic key-value organization:
Just use filesystem files to keep the data

Text format: key, attribute, BL-TEXT

234332234, ["date":"23/6/2016", "title":"Angry Ronaldo throws microphone to lake", "text":"lore ipsum lore lore...;"]

Binary format:

234332234, BinObj: #124332

Parallel Query Processing

Architectural Parallelism

shared-memory (e.g. cores, processors), threading
shared-nothing (nodes with PU, MEM, DISK)
shared-disks (multiple nodes access disks)

Query Parallelism

inter-query: Horizontal Vertical or pipelined

Parallelizing Processing

to run it n times faster ... “divide to conquer”

• Partition large fact
• Replicate small dimensions
• Hope for linear speedup

• Very little data exchange between nodes

No-SQL Data Stores

Key-value data stores, Document stores, etc

Do not oblige fixed schemas
Many allow data to be directly in text or xml files
There is frequently no complex loading, compression, opt.
No Joins and no other complex relational algebra operations
Mostly PUT, GET, FILTER AND SCAN

Are usually massively parallelizable

Parallelizing Query Processing

SUM(x) over FACT, dims GROUP BY dims

Ex: Show sales per brand per month for the whole last year

SUM(x) over 1/n of data

SUM(x) over 1/n of data

SUM(x) over 1/n of data

Presentation
Pedro Furtado, U. Coimbra, Portugal
Operations Parallelism - Join

Conceptually, every Join is:
For each tuple from A
For each tuple from B
if (A.a == B.b)
result = result + A.a | B.b

WITHOUT re-SHUFFLE:
Result should be:
A(1, …), 10
A(2, …), 1
A(3, …), 10

Result is:
A(1, …), 10
A(2, …), 1
A(3, …), 10

Join Processing with Repartitioning

Partitioned Join (NOT ALWAYS POSSIBLE)
each node must have same values of join attribute

Repartition Join -> lots of overhead, SHUFFLE
nodes do not have same values of join attribute
assign join attribute hash-intervals to each node

exchange ON-THE-FLY rows between nodes for same hash

NOW JOIN

Broadcast Join -> lots of overhead, SHUFFLE
nodes do not have same values of join attribute
broadcast ON-THE-FLY full content of one of data sets to all nodes

Avoid Join Problem => partition one only

Can we place the data initially so that joins are fastest to avoid data shipment on-the-fly on every query

You MUST PARTITION every big table at PLACEMENT time...

HOW TO PARTITION?

Workload-based Partitioning

WKLOAD-BASED PARTITIONING
Join frequency, table sizes, intermediate results...

Proposed Solution Algorithm

“Very Small” Dimensions
Replicate = BROADCAST, COPY TO ALL NODES ONCE

Non-small “pure” Dimensions
Hash-Partition by PRIMARY KEY (JOINS)
Large relations / Facts

Find key that minimizes repartitioning costs
Hash-partition by it


**WK-LOAD-BASED PARTITIONING**

Relations, Query Joins, Query Intermediate Result Sizes, ...

Minimize the number and cost of repartitioning:

- From past workload: \( \text{LINK weight} = \text{Frequency of occurrence} \times \text{size of join} \)

**Step 1:** For each relation \( R \)
- Pick the link with highest weight containing the relation
- The relation \( R \) is to be partitioned by the links’ equi-join attribute;

**Basic VS WKLOAD-based Partitioning**

<table>
<thead>
<tr>
<th>25 nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
</tr>
<tr>
<td>Q2</td>
</tr>
<tr>
<td>Q4</td>
</tr>
<tr>
<td>Q5</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Join Algorithm**

- Ship only selected rows from \( L \) …

**Merging bottleneck over aggregation**

- 10 GB, aggreg over union
- 100 MB, 100 nodes

**Massive NOSQL Parallelism**

**e.g. Cassandra in the perspective of Datastax**

- Fully distributed, no SPOF

**Cassandra in the perspective of Datastax**

- Primary key determines placement
- PK
  - jim
  - carol
  - johnny
  - suzy

- MDS Hash
  - jim: 45b239847b...
  - carol: 19899010...
  - johnny: 20978007...
  - suzy: 78942379...

MDS hash operation yields a 128-bit number for keys of any size.
Cassandra in the perspective of Datastax

Modern Scalable Platforms not only DATA STORE, also process

What is HDFS?
What is Hadoop?
What is Map-Reduce?
What is Hbase?

To view the contents of 'product':

```
> hbase > create 'product', 'characteristics'
> hbase > put 'product', 'characteristics:size', '100'
```

```
> hbase > scan 'product'
```

Map-Reduce
Way to parallelize computation
1. THINK OF AN ALGORITHM
2. PLACE IT UNDER M/R MODEL
Input= FLAT file organized as (Key, value)

Map: input=set of ((key,),value)
for each line or item of (value)
do whatever
get new k, value from it
emit(k, value)

Reduce: for each pair(k, value)
merge somehow
get new k, value from it
emit(k, value)

Map and Reduce

Hadoop and Map-Reduce: specify algs

Word Count = count the number of times each word appears in a document

```
//Pseudo-code for "word counting"
map(String key, String value):
// key: document name,
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int word_count = 0;
for each v in values:
word_count += ParseInt(v);
Emit(key, AsString(word_count));
```

Modern Scalability Platforms (hadoop + MR)
Spark

Pedro Furtado
Sixth European Business Intelligence & Big Data Summer School (eBISS 2016)

The Platform and Architecture

Hadoop
Distributed data infrastructure: It distributes massive data collections across multiple nodes within a cluster of commodity servers;
It also indexes and keeps track of that data, enabling big-data processing and analytics;
Hadoop = Hadoop Distributed File System + MapReduce + ...

Spark
Data-processing tool that operates on those distributed data collections; it doesn't do distributed storage.
Spark = Data processing framework, needs a file management system, preferably distributed for scalability (e.g. HDFS or cloud)

Scala (according to wikipedia)
Scala is a programming language for general software applications.
Scala has full support for functional programming and a very strong static type system.
Designed to be concise,[8] many of Scala’s design decisions were inspired by criticism of the shortcomings of Java.

Resilient Distributed Dataset RDD

“Resilient Distributed Datasets (RDDs) are a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

transformations - lazy operations that return another RDD.
actions - operations that trigger computation and return values.

Represents an immutable, partitioned collection of elements that can be operated on in parallel


https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-rdd-transformations.html
Resilient Distributed Dataset RDD

In-Memory
- Data inside RDD is stored in memory as much (size) and long (time) as possible.
- Immutable or Read-Only
- Does not change once created and can only be transformed using transformations to new RDDs.
- Lazy evaluated
- Data inside RDD is not available or transformed until an action is executed that triggers the execution.
- Cacheable
- You can hold all the data in a persistent “storage” like memory (default) and the most preferred or disk (the least preferred due to access speed).
- Parallel
- Process data in parallel.
- Typed
- Values in a RDD have types, e.g. RDD<Long> or RDD<Int, String>.
- Partitioned
- The data inside a RDD is partitioned (split into partitions) and then distributed across nodes in a cluster (one partition per JVM that may or may not correspond to a single node).

RDD Persistence

Each node stores its partitions in memory for re-use next.

Caching is a key tool for iterative algorithms and fast interactive use.

```scala
cache() -> stores in-memory StorageLevel.MEMORY_ONLY
persist() -> MEMORY_ONLY, MEMORY_AND_DISK,
MEMORY_ONLY_SER (more space efficient),
MEMORY_AND_DISK_SER, DISK_ONLY,
OFF_HEAP
```

Shuffle

The under-the-hood rehashing operations

Involves disk I/O, data serialization, and network I/O.

Internally, outputs from individual map tasks are kept in memory until they can’t fit.

When data does not fit in memory Spark will spill these tables to disk (disk I/O, serialization, garbage collection).

Shuffle operations can consume significant amounts of heap memory.

Shuffle also generates a large number of intermediate files on disk.

Shuffle behavior can be configured.

Broadcast Variables

Give every node a copy of a dataset.

Spark uses efficient broadcast algorithms to reduce communication costs.

Spark already automatically broadcasts the common data needed by tasks within each stage.

Explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data.

```scala
val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

Lookup tables and SMALL tables that are joined frequently.
Spark: Examples

Easy to Code (Scala, Python, Java), easy to use

Word count in Spark’s Python API

```python
from pyspark import SparkContext

sc = SparkContext()

lines = sc.textFile("hdfs://...")
word_count = lines.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

Count number of lines with word ‘Spark’

```scala
scala> val textFile = sc.textFile("README.md")
scala> textFile.filter(line => line.contains("Spark")).count()
```

Spark: Processing Data and Using Mlib

Read the file, transforming the lines into float fields

```scala
inp_file = sc.textFile("hdfs://...")
car_rdd = inp_file.map(lambda line: [float(y) for y in line.split()])
car_rdd.count
```

Calculate statistics for the minimum, maximum and average of each attribute

```scala
summary = Statistics.colStats(car_rdd)
```

president_wordings = Statistics.corr(car_rdd, method="pearson")

```scala
president_wordings
```

pySpark: Processing Data and Using Mlib

Read the file, transforming the lines into float fields

```scala
import csv
inp_file = spark.sparkContext.textFile("hdfs://...")
car_rdd = inp_file.map(lambda line: [float(y) for y in line.split()])
car_rdd.count
```

Calculate statistics for the minimum, maximum and average of each attribute

```scala
summary = Statistics.colStats(car_rdd)
```

president_wordings = Statistics.corr(car_rdd, method="pearson")

```scala
president_wordings
```

HADOOP YARN: Cluster Resource Manager

Yet another Resource Negotiator: 2nd-gen Hadoop, YARN is now a large-scale, distributed operating system for big data applications – manages the resources, with multiple request submitters and so on.

HDFS (Hadoop Distributed File System)

Presidents Wordings (1)

Get the vector of words designated as President BO (Barack Obama). Then show the number of words and the frequency of each word.

```scala
from operator import add
lines = sc.textFile("2009-2014.csv")
words = lines.flatMap(words)
word_count_b = lines.flatMap(lambda line: line.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)
```

pySpark

Use corr function from pyspark.mllib.stat statistics to study the correlation between vehicle power (hp - horse power) and vehicle weight (weight). Get the correlation based on two alternative methods, “pearson” and “spearman”. What do you conclude about the degree and form of correlation?

```python
hp = car_rdd.map(lambda x: x[2])
weight = car_rdd.map(lambda x: x[10])
weight.foreach(lambda a: print("%4.4f "% a))
print("%4.4f %" % Statistics.corr(hp, weight, method=pearson))
```

pySpark

Create code to analyze the speeches of 5 US presidents to determine the most relevant issues and challenges at the time each of them was president.

The datasets are the speeches of George Washington (GW), Abe Lincoln (AL), Franklin Delano Roosevelt (FDR), Kennedy (JFK), Bill Clinton (BC), GW Bush (GB) & Barack Obama (BO)

PDA (Plan of Action)

Read the 7 data sets to 7 RDDs

Create the word vectors

Transform into frequency of words vector

Remove common stop words

Inspect the keywords to get top n for each president

Calculate the differences between top wording of two presidents

http://spark.apache.org/docs/latest/quick-start.html
Presidents Wordings (2)

Detect the top differences between speech of BO and GWB. Do this in two ways:
1. show the top 5 words of each,
2. compute and print the difference between those speeches.

```python
word_count_bo_clean.take(10)
word_count_gwb_clean.take(10)
```
Problem of DFs

No Type Safety, types are "lost"

scala> :type df.collect
Array[org.apache.spark.sql.Row]

Rows are defined as: Row extends Serializable

Mapping it back to something useful is ugly and error-prone:

df.collect().map { row =>
  val project = row(0).asInstanceOf[String]  // Yuck.
}

Datasets

It has a type...

read a JSON and say you want to see it as Person
names must match

val df = sqlContext.read.json("people.json")

// Convert the data to a domain object.
// case class Person(name: String, age: Long)
val ds: Dataset[Person] = df.as[Person]

CONCLUSION: DataFrame is a DataSet with a "useless" datatype ROW

RDDs, DataFrames and Datasets in Apache Spark

Brian Clapper, in NE Scala 2016

Datasets VS DataFrames and RDDs

RDDs:

val lines = sc.textFile("hdfs://path/to/some/ebook.txt")
val words = lines.flatMap(_.split("\s+"))
val counts = words.groupBy(_.toLowerCase).

DataSets:

Easier to understand, look at blue code

val lines = sqlContext.read.text("hdfs://path/to/some/ebook.txt").as[String]
val words = lines.flatMap(_.split("\s+"))
val counts = words.groupByKey(_.toLowerCase).count()

Datasets – Memory and Performance

Spark has to serialize data... a lot send data across the network, e.g.
group by key=SHUFFLE=SERIALIZE dump to disk => serialize + deserialize later

Speed

Caching is important

if you use something more than once, cache it!

Use broadcast variables liberally

Only serialized once
Available to all executors
Extremely fast lookups

Distributed memory maps well parallelized

Nr of partitions should be much larger than nr of nodes
Workload-based partitioning would be great here!

Spark Scalability Experiments with TPCH

R. Rei, P. Furtado FCTUC Coimbra, Portugal
**TPC-H on Spark/Hive**

Create a HiveContext using the SparkContext as input:

```
vil hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

SQL computations in Spark can be done by applying transformations and actions on Dataframes. The HiveContext object is used to create Dataframes via reading data stored in Hive tables. The method `sql` next is a `TRANSFORMATION` (lazy evaluation, not run immediately):

```
vil DF_queryX = hiveContext.sql("**")
```

Lazy evaluation using ACTION `show` on DataFrame:

```
scaleDF_queryX.show(10, false)
```

---

**Processing FlowChart**

Show ACTION

Spark passes the query in method “sql” to optimizer

Optimizer turns query into Spark code

The optimizer = Catalyst

- compiles the operations applied to DataFrame
- Applies optimizations
- Creates a physical plan

Computation is performed by the Executors residing in Workers processes in slaves.

---

**Experimental Setup**


R. Rei, P. Furtado ICTUC Coimbra, Portugal

1 node:
- 12 cores, 32GB de RAM, 70GB disk

3 nodes:
- each with 12 cores, 32GB de RAM and 70GB disk

OS Ubuntu 14.04 LTS versión 64 bits.

Hadoop version 2.7.1, Hive versión 1.2.1, a Spark versión 1.6.1

MySQL as Metastore of Hive.

TPC-H data by ORGEN with scale factor 10GB.

Stored as Hive tables in HDFS.

---

**Hive on bigger data**

(Rafael, Furtado, Bernardos, 2014)

**Streaming, Lambda Architecture**

Pedro Furtado
Sixth European Business Intelligence & Big Data Summer School (eBISS 2016)
Lambda Architecture, Streaming

- Analytics in realtime
  - e.g. recommendation engine which recommends in realtime recomputes recommendation model very often to reflect changes
  - e.g. pricing model changes in context, competition, sales, reflected ASAP
- Requires operating on streams of data rebuilding the model deploy model that runs quickly scale, tolerate faults

Lambda Architecture

Batch layer allows you to always go back to correct state, because you have everything always
Since you have all data in the batch layer, you can create new models create an updated model correct a model incorporate new things in a model incorporate new data to rebuild the model

Batch layer + streaming layer allows you to answer all types of queries, forget about OLTP + OLAP duality form the point of view of using the data

Lambda Architecture with SPARK

Apache Spark
you have Spark (batch layer) and Spark Streaming (streaming data) spark streaming is integrated into spark same programming and model as spark, same code runs, completely integrated can use any input data (streams, logs, files, from ports, subscribe to kafka) has rich machine learning library has a lot other capabilities is totally scalable

Apache Kafka
pub/sub messaging system

You can build Lambda with spark alone!!
Kafka example - Producer

```
Kafka example - Consumer

Discretized Streams (Dstreams)

Spark Streaming

Spark Streaming
```
Spark Streaming – Wordcount app

Create a DStream that represents streaming data from a TCP source

```scala
(valssc = newStreamingContext(conf, Seconds(1)))
```

```python
# Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
```

Spark Streaming – Wordcount app

Split the lines by space characters into words
Each line will be split into multiple words and the stream of words is represented as the words Dstream

```scala
val words = lines.flatMap(_.split(" "))
```

```python
val words = lines.flatMap(_.split(" "))
```

Spark Streaming – Wordcount app

To start the processing after all the transformations have been setup, call

```scala
ssc.start() // Start the computation
ssc.awaitTermination() // Wait for the computation to terminate
```

```python
valssc = newStreamingContext(conf, Seconds(1))
```

Spark Streaming – Wordcount app

Need to feed port with phrases...

run Netcat (a small utility found in most Unix-like systems) as a data server

```bash
$ nc -lk 9999
```

in a different terminal, you can start the example
Qualities needed

- Scalability
- Partition
- Replicate for fault tolerance
- Share-Nothing
- Asynchronous Message Passing
- Parallelism
- Isolation
- Data Locality
- Location independence

Kafka:
- High-throughput, distributed messaging
- Supports massive # consumers
- Partition on cluster
- Auto-recovery node fail

Basic Spark + Cassandra

Retrieve from commits table: select and where is pushed to Cassandra for efficiency

```scala
val rdd = sc.cassandraTable("github", "commits")
.withConsistencyLevel(ConsistencyLevel.ONE)
.select("user", "count", "year", "month")
.where("commits >= ? and year = ", 1000, 2015)
```

Now retrieve from commits_aggregate into a streaming rdd:

```scala
val rdd = sc.cassandraTable("MonthlyCommits", "github", "commits_aggregate")
.withConsistencyLevel(ConsistencyLevel.ONE)
.select("user", "project_name", "year", "month")
```

Kafka Streaming Word Count

As soon as data starts coming into kafka topics, computation starts:

```scala
sparkConf.set("spark.cassandra.connection.host", "10.20.3.45")
val streamingContext = new StreamContext(sparkConf, Seconds(30))

// KafkaUtils is used for creating streams with input
// and output messages
val stream = KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
    streamingContext, kafkaParams, topicMap, StorageLevel.MEMORY_ONLY)
.map(_._2)
.countByValue
.saveToCassandra("my_keyspace", "wordcount")
```

Under Load...

A database engine should not STALL while processing a “normal” query.... Should it?
What if there are 1000 queries, even on smaller data?
One-query-at-a-time model is a problem

From traditional to realtime DW:
FRESHNESS

From traditional to realtime DW:
INTERACTIVE

(near)Realtime DWs
What if we need/want fresh data?
- Incorporate events into analysis quickly
- (e.g. Detect fraud or react quickly to events)

Why is it difficult in traditional DWs?
Which factors influence those near-realtime capabilities?
Are there solutions to make existing DW near-realtime?

Experimental Setup
Oracle DBMS
SSB star schema benchmark + REALTIME LOADING
The Query Workload had 13 queries
Two computers used in tests
Intel(R) Core(TM) 2 Quad 2.5GHz with 4GB of RAM
Intel(R) Core(TM) i5 3.40GHz with 16GB of RAM
Each test (15 runs, average 10) (+/-5%)

Test 1: effect of simultaneous loading on query performance
Test 2: Online loading vs Offline loading

Loading performance is severely impacted

High EFFECT!!!

Test 3: Effect on loading of increase in nr of query sessions

Effect of number of simultaneous query sessions on loading 10 rows of data (shown in seconds)

HUGE EFFECT

RealTime-Data Warehouse

Totally separate DW from RT Component

Merge DW and RT query results on-the-fly

Still need to load RT data into DW periodically, offline

RealTime-DW Realtime results

MONITOR to detect problems:

SCALE automatically:

Results – Auto-scaling
Traditional Parallel DBMS seem complex, and they can fail to run FAST...

De-normalize TOTALLY Parallelize

*(2) João Pedro Costa, J. Cecílio, P. Marques, and P. Furtado, "ONE: A Predictable and Scalable DW Model," DaWak'11

Immutable, append-only

The ONE repository is completely IMMUTABLE, append-only
There are no deletes or updates
(except for input typo undos)

ONE is just a huge denormalized log of data
ONE’s data is always correct
Querying and other data management are automatically rewritten to work on ONE

Parallel ONE (ONE-P) *[2]*

Simpler deployment
ONE relation fully partitioned among nodes
Partition size fitting node’s capabilities
No repartitioning is required

I WANT $x \times (m)$ seconds GUARANTEE, ALWAYS!!!!

Now you can finally say:

I want any query to take at most $X$
And the system will get you $X$... if you have machines for that

P-ONE does just that, because execution time is totally predictable

 Execution Time

<table>
<thead>
<tr>
<th>Blocksize</th>
<th>Tl</th>
<th>Ti</th>
<th>Tg</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>32 KB</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>64 KB</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

SPIN: But what about dealing with 1000 Concurrent Queries?
Massive Processing

Divide and Conquer on every step

Parallel Feature Extraction;

Algorithms execute over parts of the m-dim dataset;

Matrix Computations can be partitioned over nodes

---

Data mining, m-dim Datasets

1. Collect Cases
2. Extract Features
3. Create m-dim Feature Vector with Class/value
4. Train Classifier, Regressor, Recommender, ...
5. Get Features from new CASE
6. Extract Features
7. Create m-dim Feature Vector
8. Call Model to determine Output: Class, value, choice

Titanic (binary classification)

We are given details of Titanic passengers (e.g. name, gender, fare, cabin)

We are told whether the person survived the Titanic disaster.

Build a Model that can predict if any passenger is likely to survive.

---

Ex: Dataset Diabetes and Obesity

Adult, 57, bad food quality, 3days/week 30 mins exercise... is he in risk of obesity or diabetes?

---

SPIN: Providing Scalable concurrent query loads

Combine similar predicates

Share intermediate results

q_0 = SUM(q) (value (1)
q_1 = SUM(q) (value (2))
q_2 = SUM(q) (value (3))

---

Analytics

Pedro Furtado

Sixth European Business Intelligence & Big Data Summer School (eBISS 2016)
Tintanic: load and analyze data

Recommendation Example

Load the files

Transform fields

Create DS train, test, validation
Train model – Alternating Least Squares

Train the model using 20 iterations, with variable rank = 10.

```java
from pyspark.mllib.recommendation import ALS

rank = 10
numIterations = 20
model = ALS.train(train_data, rank, numIterations)
```

Evaluate model

Evaluate the model in the training data. Show the first recommendation, the first line of test data and the first real rating (test).

```java
# Evaluate model on test data
predictions = model.predictAll(testdata).map(lambda (id, r): ((id, r[1]), r[2]))
predictions.map(lambda (uid, pid), actual: ((uid, pid), actual, model.predict(uid, pid))).collect()
```

Practical Cases: Synset

**Practical Case 1: Synset “dish” of Imagenet**

Use Machine Learning algorithms for annotation and classification of images

- Multi-class classifier (decision trees, naïve bayes, random forests)
- Dishes Synset contains 313 classes of recipes
- We had to decrease the nr of classes in practical testbed (10 to 50)
- Later we will do it for all classes
- We added a customized SVM algorithm (two classes 1/all)
- We optimized pre-processing + analyzed partitioning scalability
- Compared accuracy and runtime

**Future work (very much looking forward to it)**

Deep Learning approach


UC-DIEI TechReport1010

Summary of Steps

**Summary of Steps**


UC-DIEI TechReport1010

Proposed Roadmap

**Proposed Roadmap**

Pre-processing

- Reduce images to GrayScale (python cv2 binding to OpenCV)
- Transfer all images to HDFS ‘/synsets’-‘/usabhi, etc

Feature extraction

- SIFT features -> keypoints->bagofwords->k-means->pooling
- Using OpenCV-vgg-Spark

Creation of train and test datasets

- 10 to 15% of images = test
- Train Spark classifier OFF-THE-SHELF


UC-DIEI TechReport1010

Results SS

**Load time to HDFS:**

<table>
<thead>
<tr>
<th>nr Classes</th>
<th>nr Images</th>
<th>MB</th>
<th>load time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>624</td>
<td>873</td>
<td>252.3</td>
</tr>
<tr>
<td>50</td>
<td>29326</td>
<td>3500</td>
<td>492.61</td>
</tr>
</tbody>
</table>

Note: one of the machines is simultaneously master and slave


UC-DIEI TechReport1010
Some References
Saha, Costa; Pedro Marques, José Cecilio, Pedro Furtado, "Providing timely results with an Elastic Parallel DW", in The 20th International Symposium on Methodologies for Intelligent Systems, 6-7 December 2012, Lisbon.
Furtado, Pedro; Costa, João Pedro; Furtado, Pedro, in Dawak 2013.
Nickerson Ferreira, Pedro Furtado: "RTDW", in IDEAS 2013.

Take-away:
- Concepts are more useful/relevant than systems

Thank you all!
Pedro Furtado
pnf@dei.uc.pt

The END...questions?

http://eden.dei.uc.pt/~pnf/

Pedro Furtado, U. Coimbra, Portugal