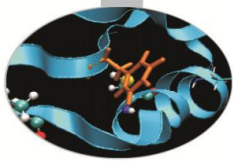
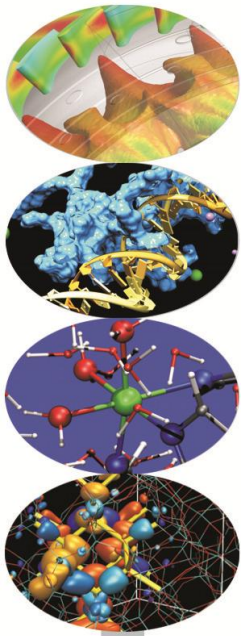
CINECA SCAI
SuperComputing Applications and Innovation



Machine learning with Spark



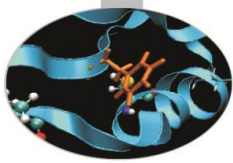
Giorgio Pedrazzi

School of Scientific Data Analytics and Visualisation

Bologna, 21/06/2016

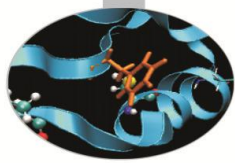
CINECA

Agenda



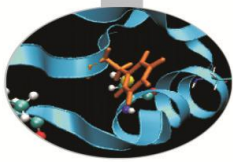
- An introduction to Spark
- Examples with Spark MLlib in Scala and Python
- Unsupervised learning: Clustering
  - Distance measures
  - K-means
  - Clustering validation
- Supervised learning: Classification
  - Training and test
  - Evaluation metrics
  - Decision tree
  - Naïve Bayes

Spark Lightning-fast cluster computing



- Spark is an open source **in-memory computing** framework for **distributed** data processing and **iterative** analysis on **massive** data volumes providing the ability to develop applications in Java, Scala, Python and R.
- It has an advanced DAG execution engine that supports cyclic data flow and in-memory computing.
- It has numerous advantages over Hadoop's MapReduce execution engine, in both the speed with which it carries out batch processing jobs and the wider range of computing workloads it can handle.
- It integrates with the Hadoop ecosystem and data sources (HDFS, Amazon S3, Hive, HBase, Cassandra, etc.)

Spark Ecosystem

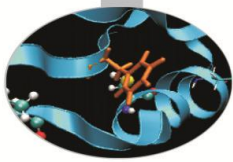


Engineers use Spark's programming API to develop systems that implement business use cases.

Data Scientists use Spark for their ad-hoc analysis that give them results immediately. Data Scientists use SQL, statistics and machine learning.

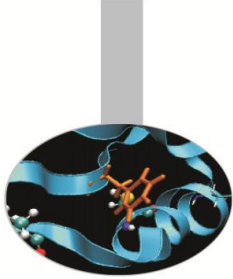
Scala Java python R

Spark Ecosystem



- **Spark SQL** is designed to work with the Spark via SQL and HiveQL. It allows developers to intermix SQL with Spark's programming languages.
- **Spark Streaming** provides processing of live streams of data. It also provides the same degree of fault tolerance, throughput, and scalability that the Spark Core provides.
- **MLlib** is the machine learning library that provides multiple types of machine learning algorithms. All of these algorithms are designed to scale out across the cluster as well.
- **GraphX** is a graph processing library with APIs to manipulate graphs and performing graph-parallel computations.

Running Spark

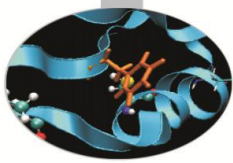


Spark runs in five modes:

- The standalone local mode, where all Spark processes are run within the same **Java Virtual Machine** (**JVM**) process*
- The standalone cluster mode, using Spark's own built-in job-scheduling framework
- Using Mesos, a popular open source cluster-computing framework
- Using YARN (commonly referred to as NextGen MapReduce), a Hadoop-related cluster-computing and resource-scheduling framework
- On a HPC traditional environment with PBS installing the spark package spark-on-hpc

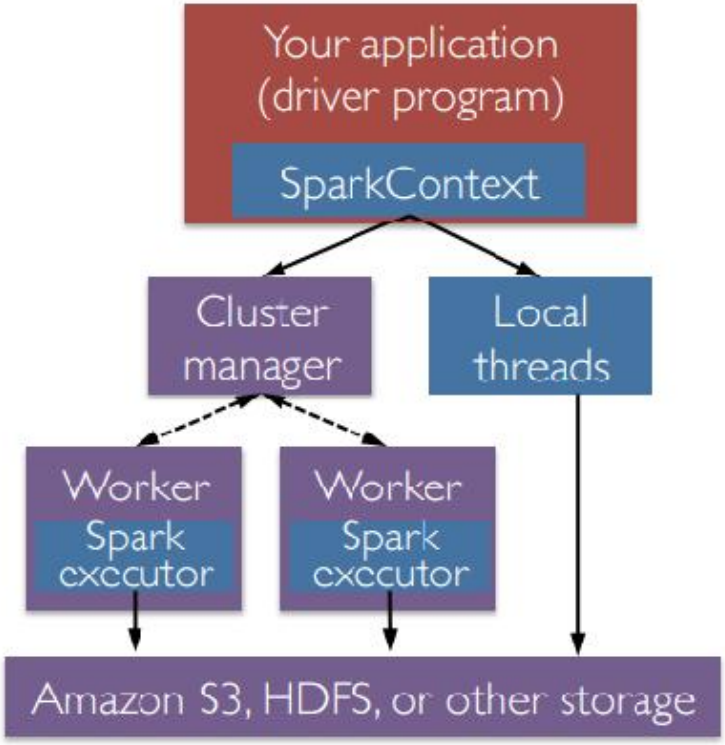
* Used during exercises with all-spark-notebook. The system is also configurable to use Mesos

Running Spark

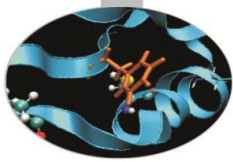


A Spark program is two programs: a driver program and a workers program

Worker programs run on cluster nodes or in local threads

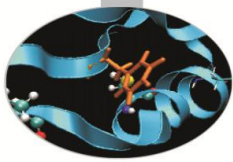


Spark shell



- The Spark shell is a tool for rapid prototyping with Spark. It helps to be familiar with Scala, but that isn't necessary. The Spark shell works with Scala, Python and R. The Spark shell allows you to interactively query and communicate with the Spark cluster. It can be invocated by these commands
- ./bin/spark-shell (Scala)
- ./bin/pyspark (Python)
- ./bin/sparkR (R)

Notebook



jupyter [201510162 R] Data Exploration Last Checkpoint: 10/29/2015 (unsaved changes)

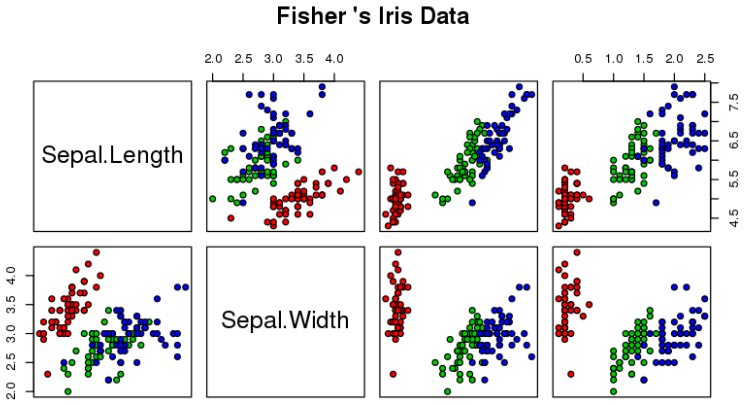
File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

The **Iris flower data set** or Fisher's Iris data set is a multivariate data set introduced by Ronald Fisher in his 1936 paper "The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis". The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.

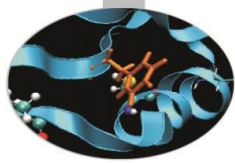
In [2]:
```
library(repr)
iris=read.csv("data/iris.csv",header = FALSE)
colnames(iris) <- c("Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species")
```

In [4]:
```
pairs(iris[1:4], main = "Fisher 's Iris Data", pch = 21, bg = c("red", "green3", "blue")[unclass(iris$Species)])
```



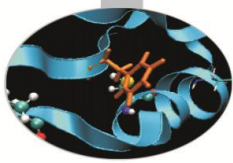
Notebook-style development provides a more exploratory way to write code than with traditional IDEs. Their interfaces are comprised of code blocks (cells), which can stand alone or act in unison. It is a discovery process, where a researcher experiments in one cell, then can continue to write code in a subsequent cell depending on results from the first. When analyzing large datasets, this *conversational* approach allows researchers to quickly discover patterns or other artifacts of the data.

Jupyter Notebook



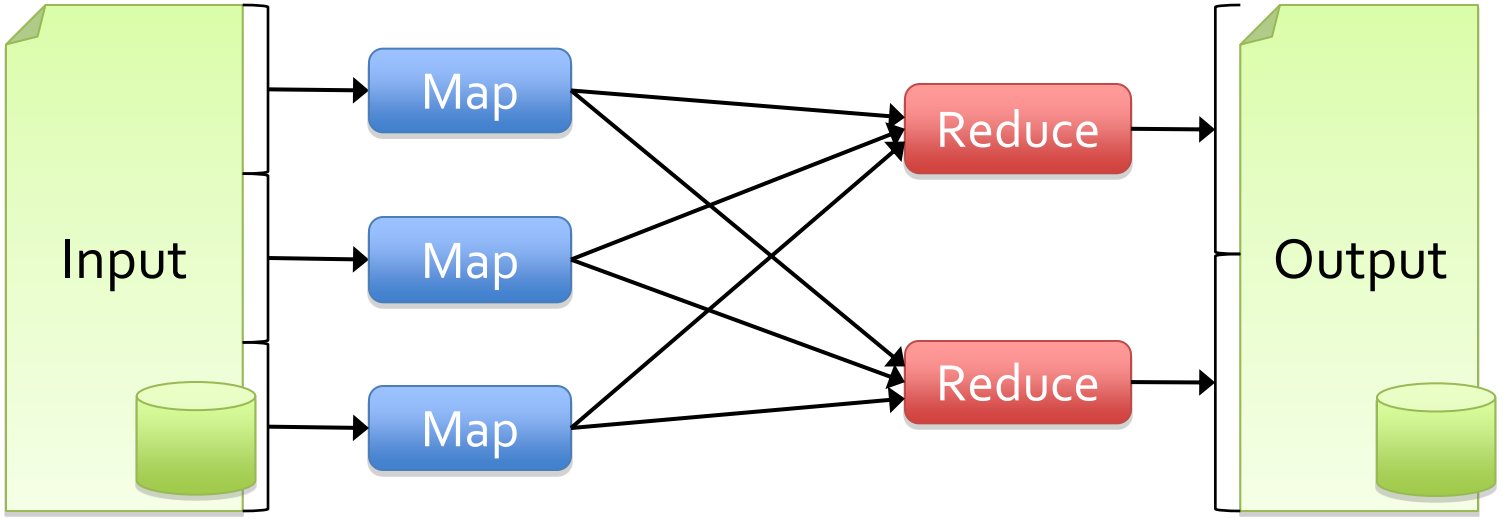
- Jupyter Notebook 4.0.x
- https://github.com/jupyter/docker-stacks/tree/master/all-spark-notebook
- Conda Python 3.x and Python 2.7.x environments
- Conda R 3.2.x environment
- Scala 2.10.x
- pyspark, pandas, matplotlib, scipy, seaborn, scikit-learn pre-installed for Python
- ggplot2, rcurl preinstalled for R
- Spark 1.5.1 for use in local mode or to connect to a cluster of Spark workers
- Mesos client 0.22 binary that can communicate with a Mesos master
- Unprivileged user jovyan (uid=1000, configurable, see options) in group users (gid=100) with ownership over /home/jovyan and /opt/conda
- tini as the container entrypoint and start-notebook.sh as the default command
- Options for HTTPS, password auth, and passwordless sudo

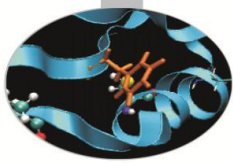
Motivation



Current popular programming models for clusters transform data flowing from stable storage to stable storage

e.g., MapReduce:

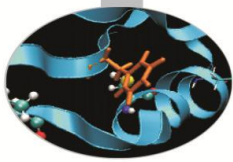




Motivation

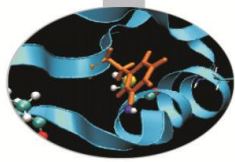
- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working sets a first-class concept to efficiently support these apps

Programming Model



- Spark introduces the concept of RDD (Resilient Distributed Dataset), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel. An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program.
- RDDs support two types of operations:
  - Transformations are operations (such as map, filter, join, union, and so on) that are performed on an RDD and which yield a new RDD containing the result.
  - Actions are operations (such as reduce, count, first, and so on) that return a value after running a computation on an RDD.

Programming Model

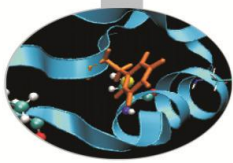


• Transformations in Spark are “lazy”, meaning that they do not compute their results right away. Instead, they just “remember” the operation to be performed and the dataset (e.g., file) to which the operation is to be performed.

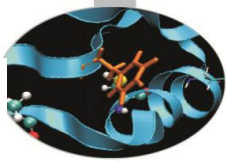
• The transformations are only actually computed when an action is called and the result is returned to the driver program. This design enables Spark to run more efficiently.

• By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist or cache method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it.

RDD Operations

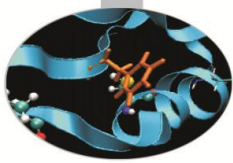


Transformations (define a new RDD)	**Parallel operations (Actions)** (return a result to driver)
map filter sample union groupByKey reduceByKey join cache …	reduce collect count save lookupKey …



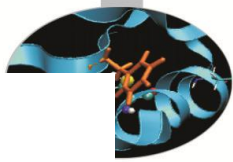
Spark Essentials: *Transformations*

transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset



Spark Essentials: *Transformations*

transformation	*description*
groupByKey([*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
cartesian(*otherDataset*)	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)



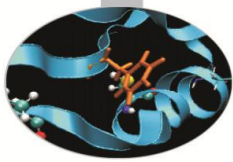
Spark Essentials: *Actions*

action	*description*
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n***)**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: *Actions*

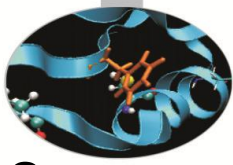
action	*description*
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

SparkSQL



- SparkSQL is a Spark component that supports querying data either via SQL or via the Hive Query Language. It originated as the Apache Hive port to run on top of Spark (in place of MapReduce) and is now integrated with the Spark stack. In addition to providing support for various data sources, it makes it possible to weave SQL queries with code transformations which results in a very powerful tool.

DataFrames



- DataFrames have been introduced in Spark 1.3 as extension to RDDs
- A DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python.
- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.
- What you can do in Spark SQL, you can do in DataFrames and vice versa.
- The DataFrame API is available in Scala, Java, Python, and R.