**Apache Spark and Python for Big Data and Machine Learning**





**Team 9:**

Khushbu Parekh

Tapadyuti Maiti

**Under the Guidance of:**

Prof. Shrikanth Krishnamurthy

**Introduction to Apache Spark**

Apache Spark is known as a fast, easy-to-use and general engine for big data processing that has built-in modules for streaming, SQL, Machine Learning (ML) and graph processing. This technology is an in-demand skill for data engineers, but also data scientists can benefit from learning Spark when doing Exploratory Data Analysis (EDA), feature extraction and ML.

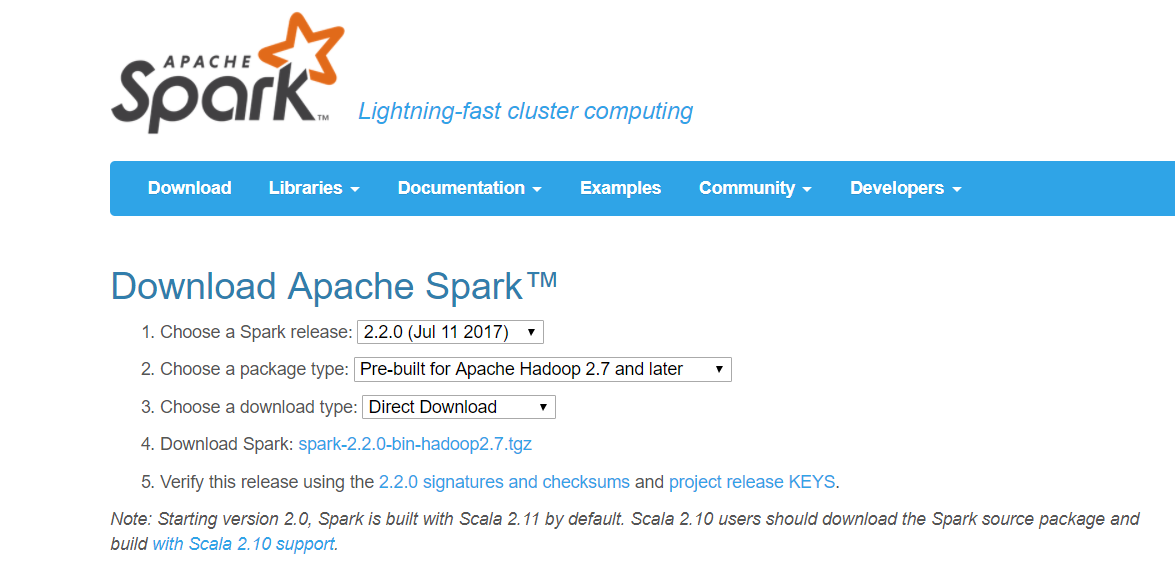
**What languages can be used?**

* Scala
* Python

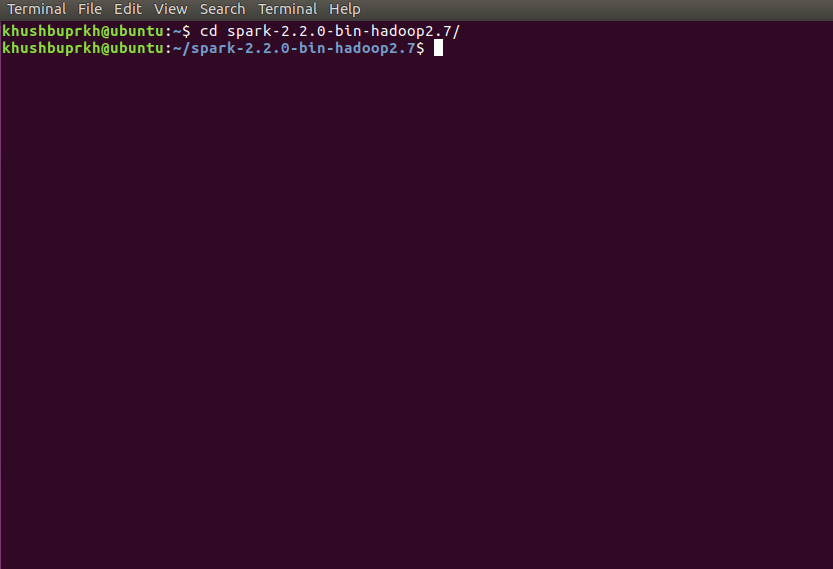
**Download and Set Up Spark for Python**

**Step1: Download spark form the link mentioned below:**

<https://spark.apache.org/downloads.html>

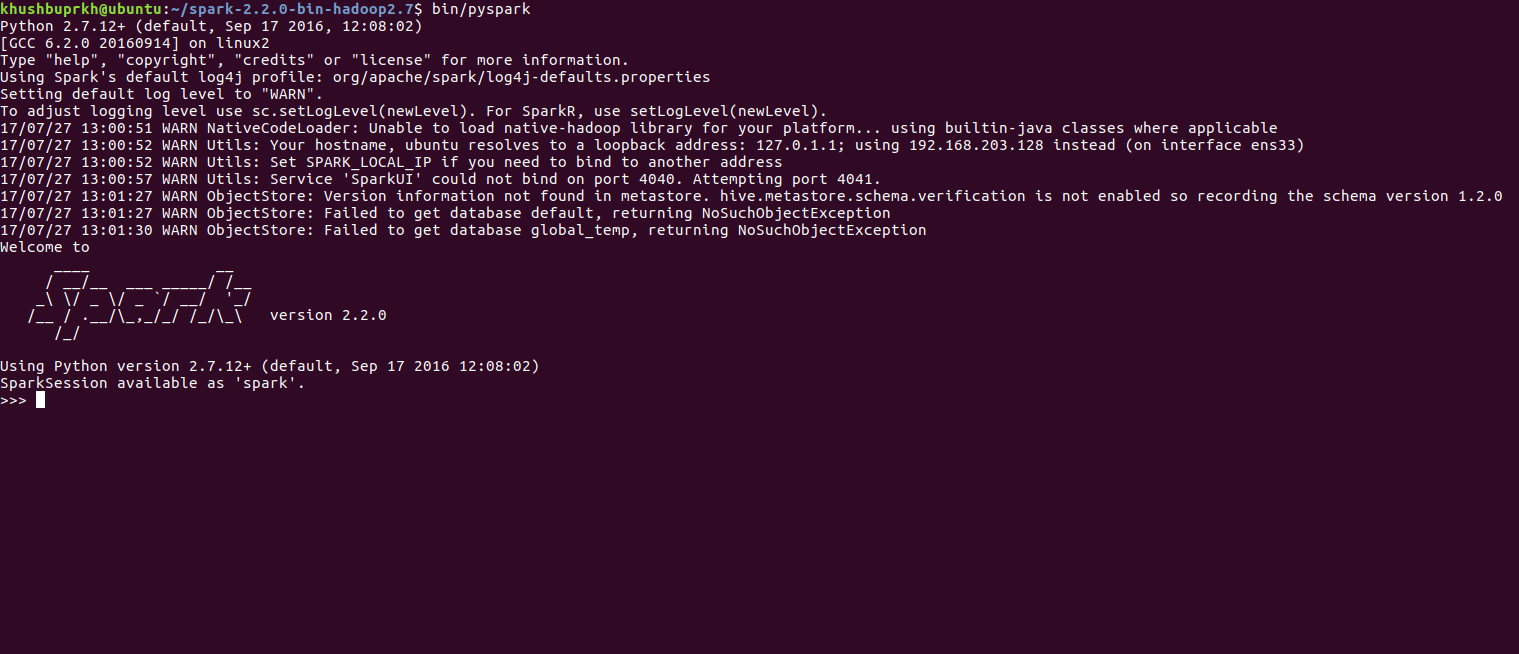


**Step2: Unzip the .tgz file downloaded and then enter the folder through command line.**

****

**Step3: To start the spark bash then enter the command “bin/pyspark”.**

****

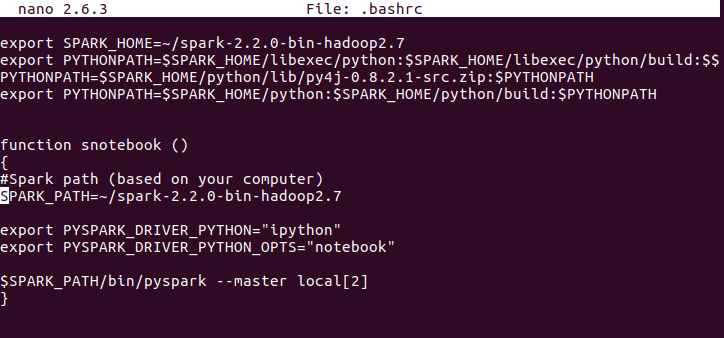
****

**Step 4: To Integrate pyspark with jupyter notebook perform the below steps:**

* Open bashrc using the command “nano .bashrc”

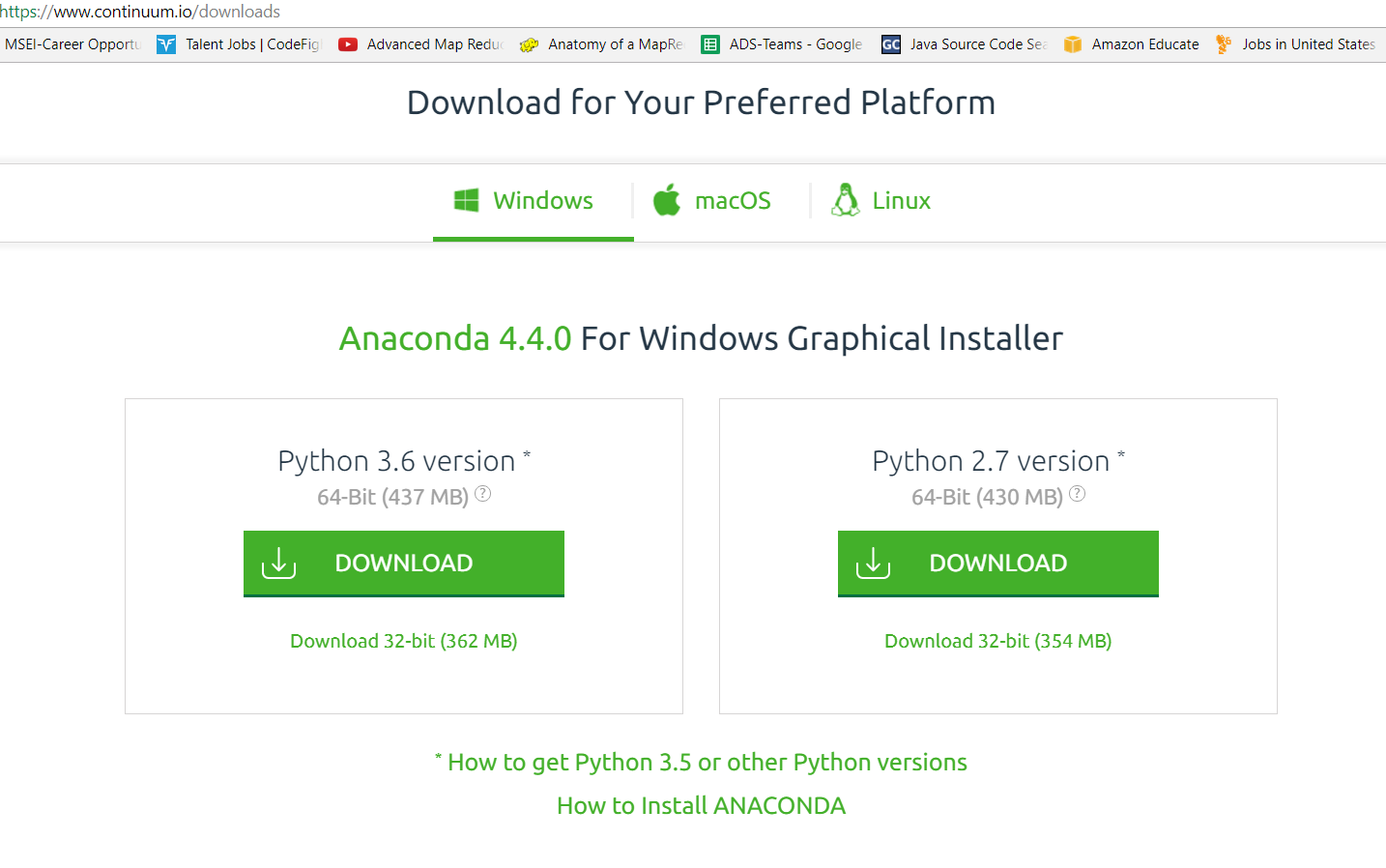
****

* Paste the below code and export statements at the end of the file:



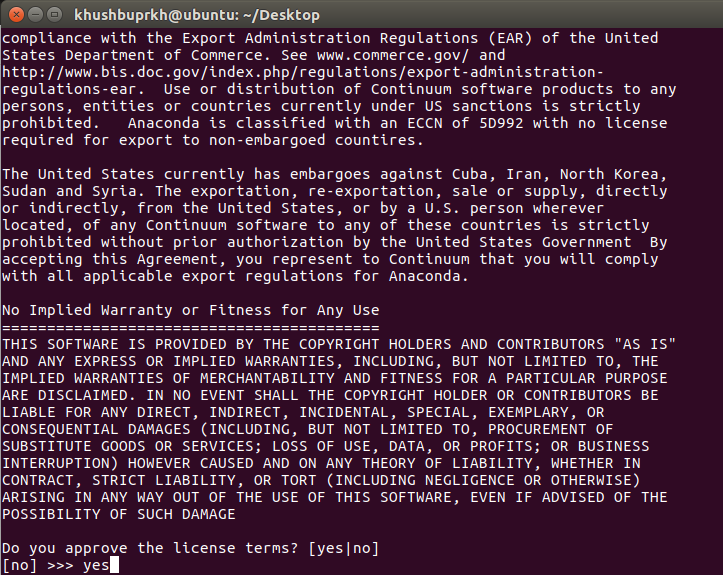
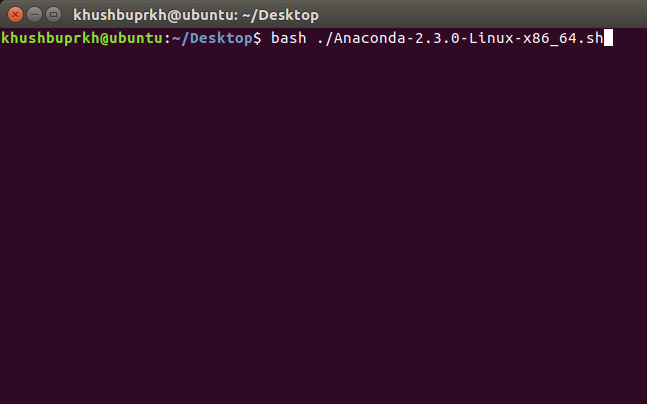
**Step 4: Download and Install Anaconda using the below link:**

<https://www.continuum.io/downloads>

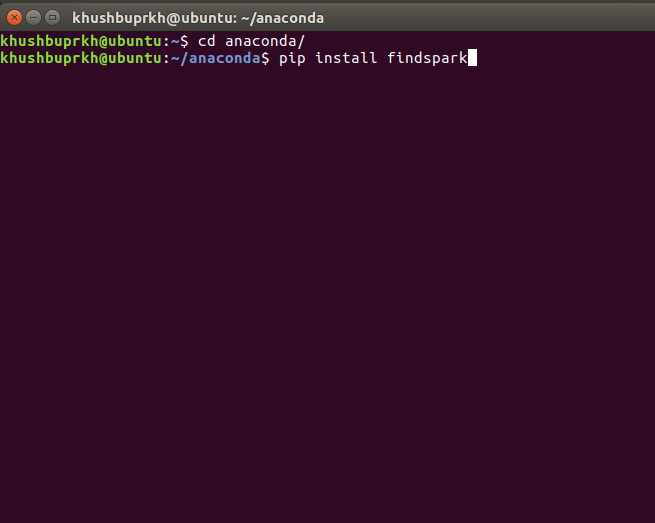


**Step 4: Install Anaconda using command:**

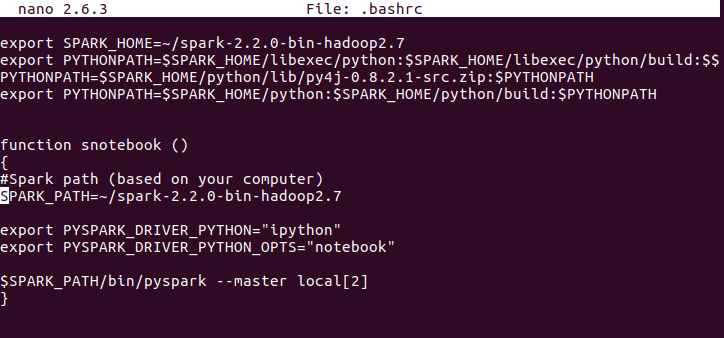
bash ./Anaconda-2.3.0-Linux-x86\_64.sh, press enter and accept the agreements



**Step 4: Get inside the anaconda folder created and get inside the bin section and execute “pip install findspark”**

****

**Step 5: To open jupyter notebook with spark enable you need to open it by typing the function created in the bashrc**





**PySpark Basics: RDD’s**

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

**There are two ways to create RDDs:**

* parallelizing an existing collection in your driver program
* referencing a dataset in an external storage system

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations.

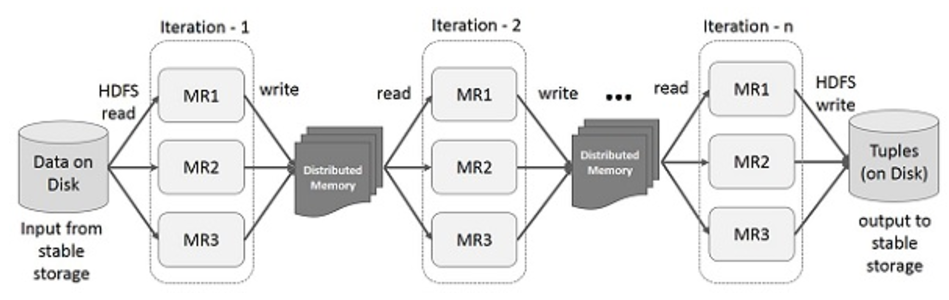
**Data Sharing using Spark RDD**

Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is Resilient Distributed Datasets (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

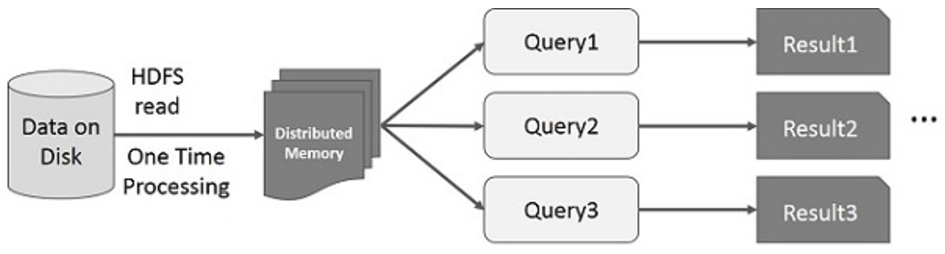
**Iterative Operations on Spark RDD**

It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.



**Interactive Operations on Spark RDD**

If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



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, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

**RDD Operations**

**There are two types of RDD operations:**

* **RDD Transformation**
  + Transformation is what you do to RDD to get another resultant RDD

e.g. : filter(), union()

* + Always returns RDD.
* **RDD Actions**
  + Actions return a result to the driver program or write it in a storage and kick off a computation.

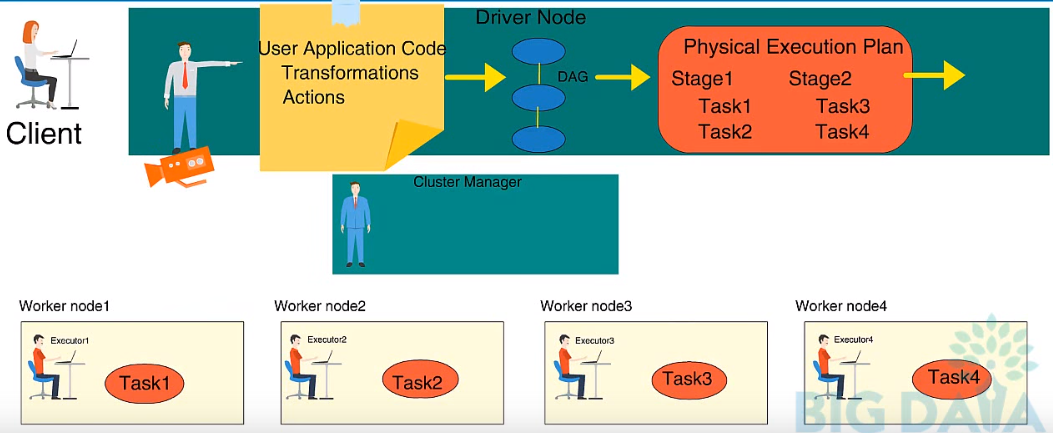
e.g. : count(), first(), collect(), take()

* + Collect function should not be used on large dataset because the entire data should fit in memory on single machine.
  + Always returns some other datatype

**Apache Spark Components**

* **Drivers :**
  + This is process where the main method of your program runs
  + Driver first coverts the user program into tasks and sends them to worker nodes.
* **Workers :**
  + These are called the compute nodes.
* **Executers :**
  + These are the JVM processes within worker nodes.
  + They run the task of the application and returns the result to the driver program.
  + They provide in-memory storage for RDD’s that are cached by user program.
* **Cluster Manager :**
  + It is an external service that launches application on a set of machines.
  + Spark comes with a built-in cluster manager called standalone cluster manager.
  + External cluster manager like: Hadoop yarn, Apache Mesos.

**Runtime Architecture of Spark Application**

****

* Apache spark uses master slave architecture.
* The client submits user application code to the Driver.
* The Driver implicitly converts user code containing transformation and actions to a logical Directed Acyclic Graph (DAG).
* At the Directed Acyclic Graph (DAG) stage it also performs optimizations such as pipelining transformations.
* Then it converts the logical DAG to a physical execution plan which sets up stages.
* After converting into physical execution plan it creates physical execution units called tasks under each stage.
* Then the tasks are bundled to be sent to the Cluster.
* Now the Driver talks to the cluster manager and negotiate for resources.
* Then the cluster manager launches Executors on the worker nodes on behalf of the Driver.
* At this point the Driver will send the task to the Executors based on the data placement.
* When the Executors start they register themselves with the Driver.
* The Driver will have complete view of the Executors.
* Then the Executors start executing the tasks assigned by the driver program.
* The Driver node schedules future tasks in appropriate locations based on the data placement.
* The user program may cache the data in certain locations using cache method or persist method.
* Driver tracks the location of the cached data to schedule future tasks that access that data.

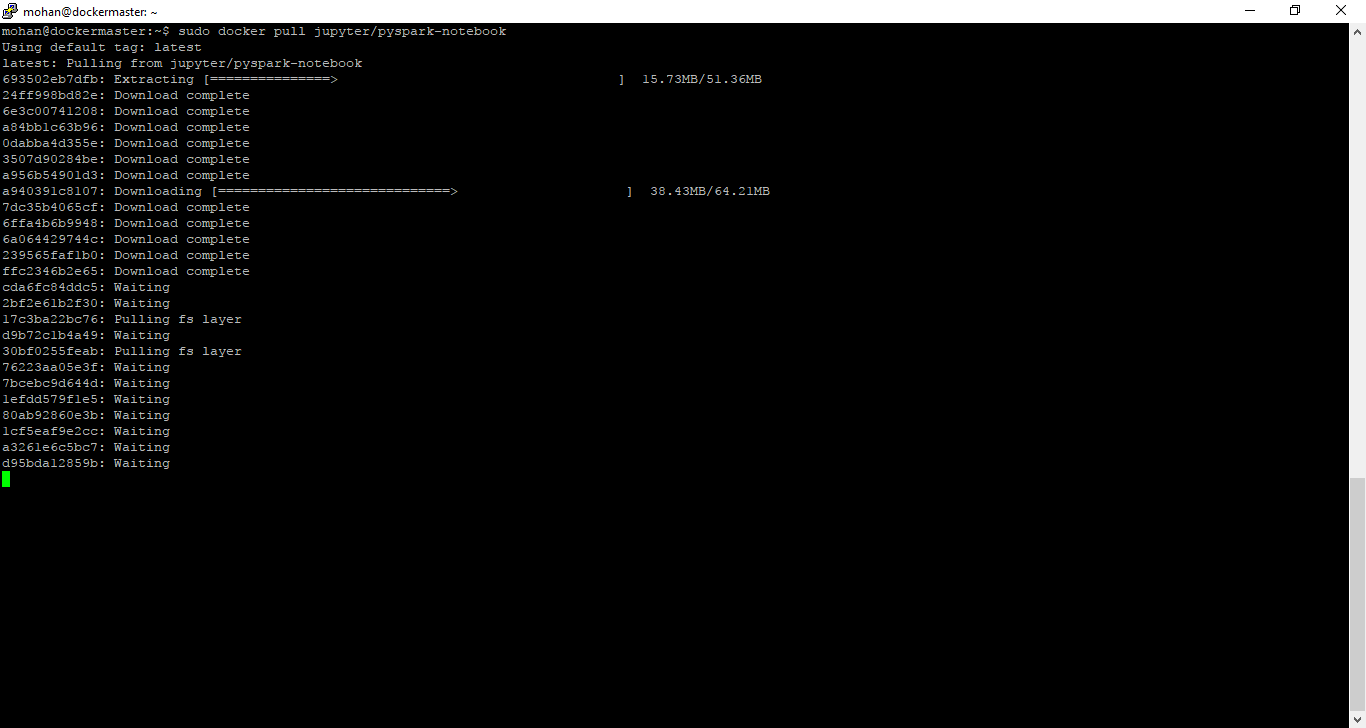
Driver exposes the running spark application through a web UI at port 4040

SparkContext.stop(), stops all the Executors and releases resources from Cluster Manager.

Running PySpark with Jupyter in Docker Containers

Pull the image from the docker by giving the following command:

**docker pull jupyter/pyspark-notebook**

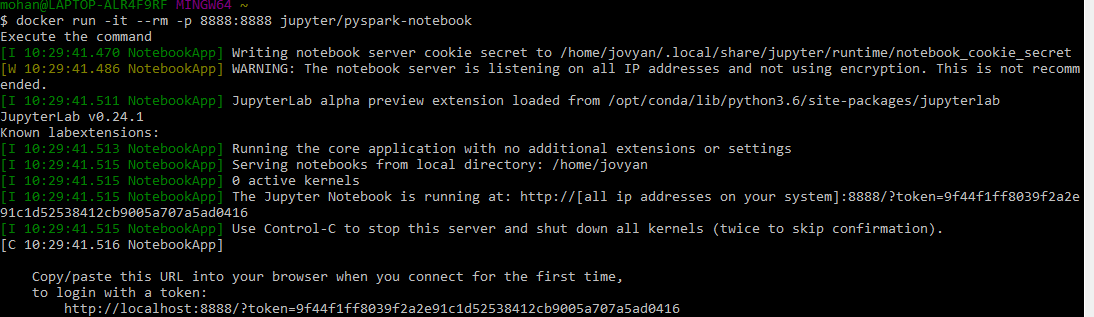


What it Gives You

* Jupyter Notebook 5.0.x
* Conda Python 3.x and Python 2.7.x environments
* pyspark, pandas, matplotlib, scipy, seaborn, scikit-learn pre-installed
* Spark 2.2.0 with Hadoop 2.7 for use in local mode or to connect to a cluster of Spark workers
* Mesos client 1.2 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](https://github.com/krallin/tini) as the container entrypoint and [start-notebook.sh](https://hub.docker.com/r/jupyter/base-notebook/start-notebook.sh) as the default command
* A [start-singleuser.sh](https://hub.docker.com/r/jupyter/base-notebook/start-singleuser.sh) script useful for running a single-user instance of the Notebook server, as required by JupyterHub
* A [start.sh](https://hub.docker.com/r/jupyter/base-notebook/start.sh) script useful for running alternative commands in the container (e.g. ipython, jupyter kernelgateway, jupyter lab)
* Options for a self-signed HTTPS certificate and passwordless sudo

Connect to the port 8888 using local host of Docker which by default 192.168.99.100 and paste the token as mentioned .

sudo docker run -it --rm -p 8888:8888 jupyter/pyspark-notebook



## Using Spark Local Mode

This configuration is nice for using Spark on small, local data.

1. Run the container as shown above.
2. Open a Python 2 or 3 notebook.
3. Create a SparkContext configured for local mode.

For example, the first few cells in the notebook might read:

import pyspark

sc = pyspark.SparkContext('local[\*]')

**SparkContext –**

Main entry point for Spark functionality. A Spark Context represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.

**Core classes:**

**pyspark.SparkContext**

Main entry point for Spark functionality.

**pyspark.RDD**

A Resilient Distributed Dataset (RDD), the basic abstraction in Spark.

**pyspark.streaming.StreamingContext**

Main entry point for Spark Streaming functionality.

**pyspark.streaming.DStream**

A Discretized Stream (DStream), the basic abstraction in Spark Streaming.

**pyspark.sql.SQLContext**

Main entry point for DataFrame and SQL functionality.

**pyspark.sql.DataFrame**

A distributed collection of data grouped into named columns.

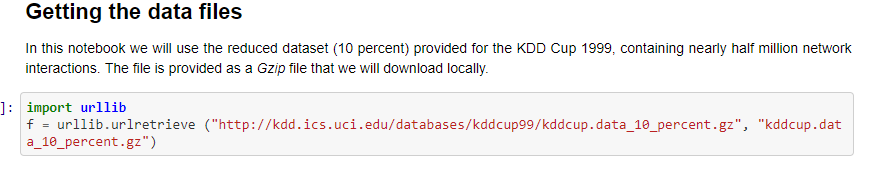
# RDD creation

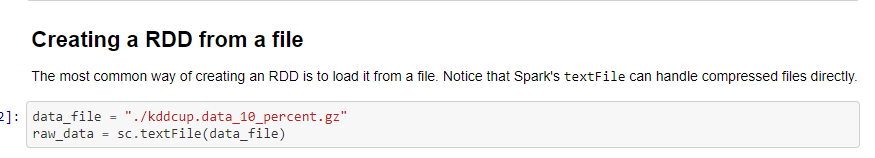
#### [Introduction to Spark with Python, by Jose A. Dianes](https://github.com/jadianes/spark-py-notebooks)

In this notebook we will introduce two different ways of getting data into the basic Spark data structure, the **Resilient Distributed Dataset** or **RDD**. An RDD is a distributed collection of elements. All work in Spark is expressed as either creating new RDDs, transforming existing RDDs, or calling actions on RDDs to compute a result. Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

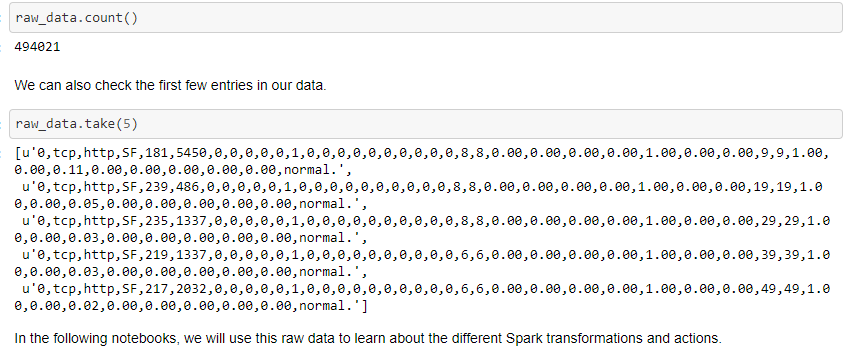
#### References

The reference book for these and other Spark related topics is Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia.

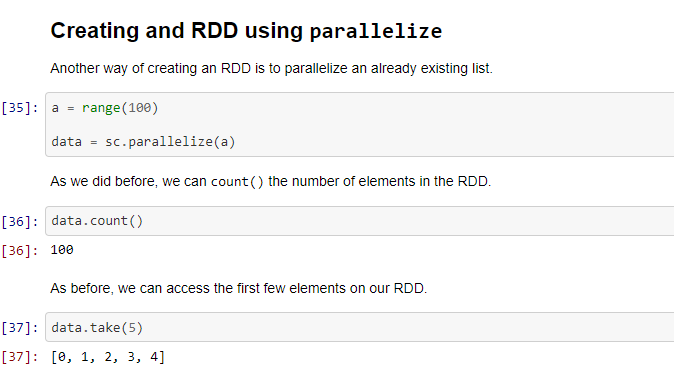




**Actions:**



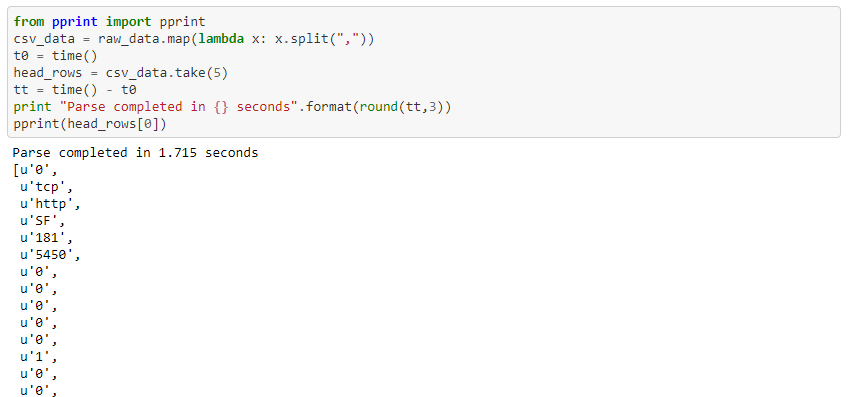
**Sample – Hello- World Example**

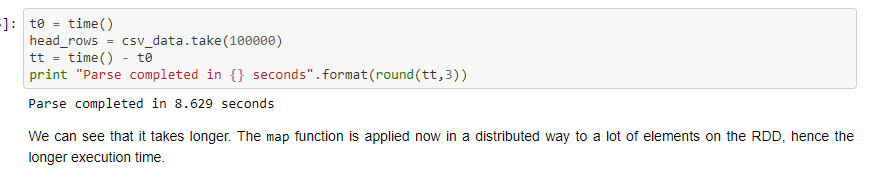


## The map transformation

By using the map transformation in Spark, we can apply a function to every element in our RDD. Python's lambdas are specially expressive for this particular.

In this case we want to read our data file as a CSV formatted one. We can do this by applying a lambda function to each element in the RDD as follows.





## The collect action[¶](https://render.githubusercontent.com/view/ipynb?commit=cd4e32e4cd933317ae668043756142f604b1cd2c&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f6a616469616e65732f737061726b2d70792d6e6f7465626f6f6b732f636434653332653463643933333331376165363638303433373536313432663630346231636432632f6e62322d7264642d6261736963732f6e62322d7264642d6261736963732e6970796e62&nwo=jadianes%2Fspark-py-notebooks&path=nb2-rdd-basics%2Fnb2-rdd-basics.ipynb&repository_id=35145949&repository_type=Repository#The-collect-action)

So far we have used the actions count and take. Another basic action we need to learn is collect. Basically it will get all the elements in the RDD into memory for us to work with them. For this reason it has to be used with care, specially when working with large RDDs.

An example using our raw data.

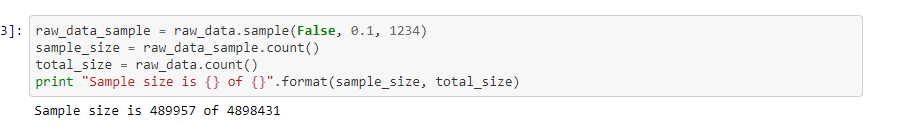
### 

## Sampling RDDs

In Spark, there are two sampling operations, the transformation sample and the action takeSample. By using a transformation we can tell Spark to apply successive transformation on a sample of a given RDD. By using an action we retrieve a given sample and we can have it in local memory to be used by any other standard library (e.g. Scikit-learn).

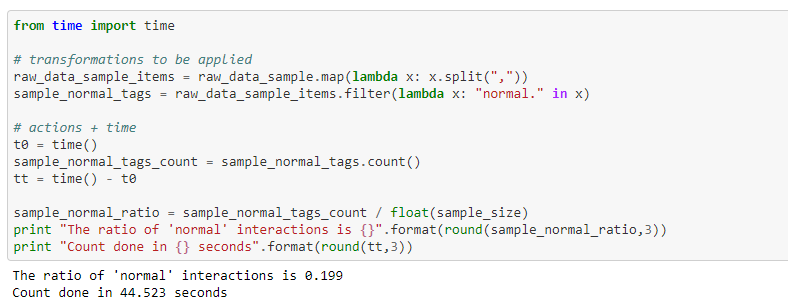
### The sample transformation

The sample transformation takes up to three parameters. First is whether the sampling is done with replacement or not. Second is the sample size as a fraction. Finally we can optionally provide a random seed.



But the power of sampling as a transformation comes from doing it as part of a sequence of additional transformations. This will show more powerful once we start doing aggregations and key-value pairs operations, and will be specially useful when using Spark's machine learning library MLlib.

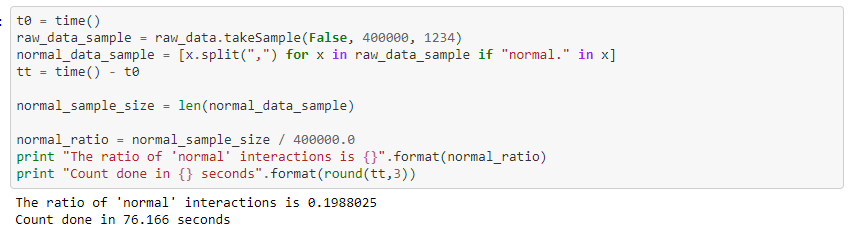
In the meantime, imagine we want to have an approximation of the proportion of normal. interactions in our dataset. We could do this by counting the total number of tags as we did in previous notebooks. However we want a quicker response and we don't need the exact answer but just an approximation. We can do it as follows.



### The takeSample action[¶](https://render.githubusercontent.com/view/ipynb?commit=cd4e32e4cd933317ae668043756142f604b1cd2c&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f6a616469616e65732f737061726b2d70792d6e6f7465626f6f6b732f636434653332653463643933333331376165363638303433373536313432663630346231636432632f6e62332d7264642d73616d706c696e672f6e62332d7264642d73616d706c696e672e6970796e62&nwo=jadianes%2Fspark-py-notebooks&path=nb3-rdd-sampling%2Fnb3-rdd-sampling.ipynb&repository_id=35145949&repository_type=Repository#The-takeSample-action)

If what we need is to grab a sample of raw data from our RDD into local memory in order to be used by other non-Spark libraries, takeSample can be used.

The syntax is very similar, but in this case we specify the number of items instead of the sample size as a fraction of the complete data size.

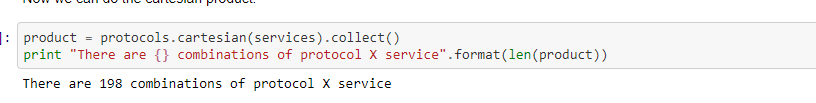


distinct and cartesian are expensive operations so they must be used with care when the operating datasets are large.

Example 1 : distinct



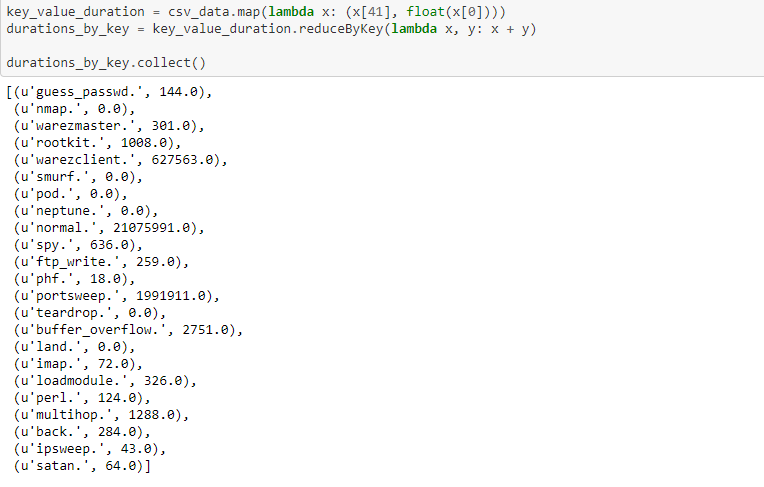
Example 2 : Cartesian



## Data aggregations with key/value pair RDDs

We can use all the transformations and actions available for normal RDDs with key/value pair RDDs. We just need to make the functions work with pair elements. Additionally, Spark provides specific functions to work with RDDs containing pair elements. They are very similar to those available for general RDDs.

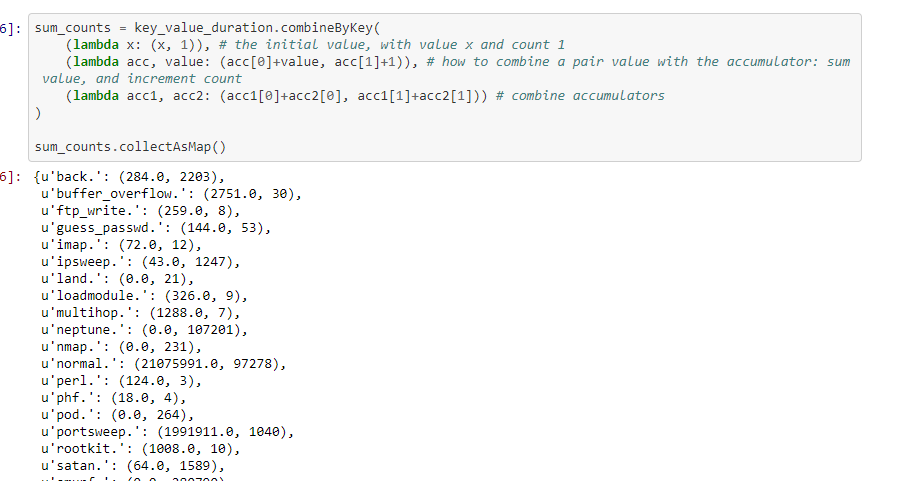
For example, we have a reduceByKey transformation that we can use as follows to calculate the total duration of each network interaction type.

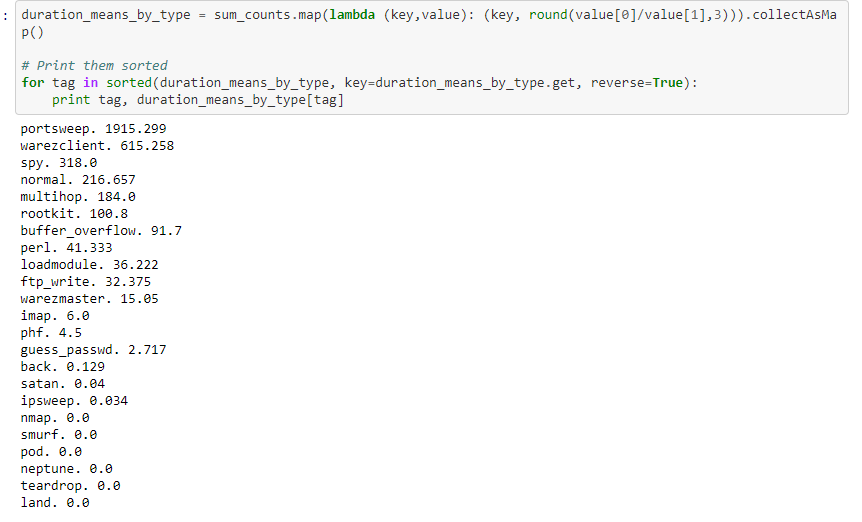


### Using combineByKey

This is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it. We can think about it as the aggregate equivalent since it allows the user to return values that are not the same type as our input data.

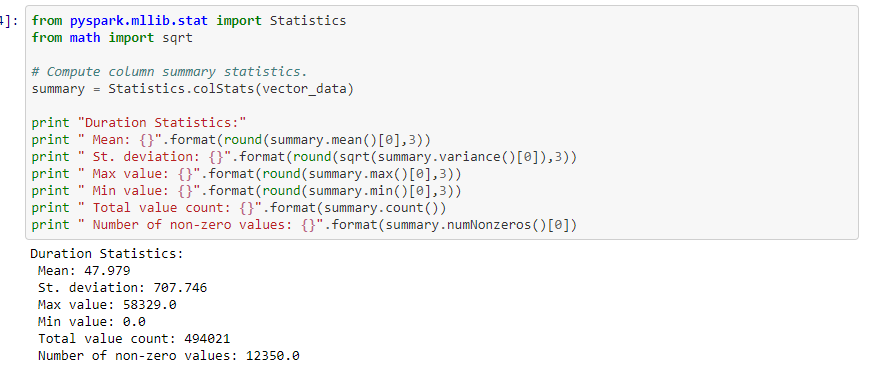
For example, we can use it to calculate per-type average durations as follows.



In Sorted Manner:   


## Summary statistics

Spark's MLlib provides column summary statistics for RDD[Vector] through the function [colStats](https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html" \l "pyspark.mllib.stat.Statistics.colStats) available in [Statistics](https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.stat.Statistics). The method returns an instance of [MultivariateStatisticalSummary](https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html" \l "pyspark.mllib.stat.MultivariateStatisticalSummary), which contains the column-wise max, min, mean, variance, and number of nonzeros, as well as the total count.



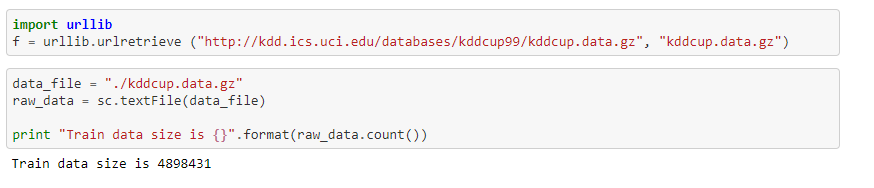
# MLlib: Classification with Logistic Regression

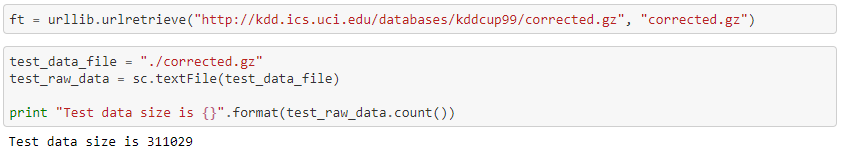
[Introduction to Spark with Python, by Jose A. Dianes](https://github.com/jadianes/spark-py-notebooks)

In this notebook we will use Spark's machine learning library [MLlib](https://spark.apache.org/docs/latest/mllib-guide.html) to build a **Logistic Regression** classifier for network attack detection. We will use the complete [KDD Cup 1999](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html) datasets in order to test Spark capabilities with large datasets.

## Getting the data and creating the RDD

As we said, this time we will use the complete dataset provided for the [KDD Cup 1999](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html), containing nearly half million network interactions. The file is provided as a Gzip file that we will download locally.



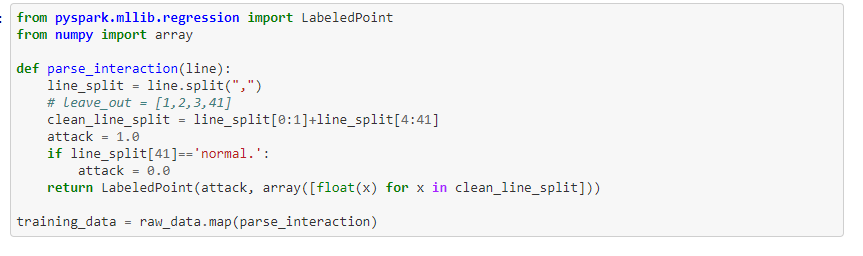


## Labeled Points

A labeled point is a local vector associated with a label/response. In [MLlib](https://spark.apache.org/docs/latest/mllib-data-types.html" \l "labeled-point), labeled points are used in supervised learning algorithms and they are stored as doubles. For binary classification, a label should be either 0 (negative) or 1 (positive).

### Preparing the training data

In our case, we are interested in detecting network attacks in general. We don't need to detect which type of attack we are dealing with. Therefore we will tag each network interaction as non attack (i.e. 'normal' tag) or attack (i.e. anything else but 'normal').

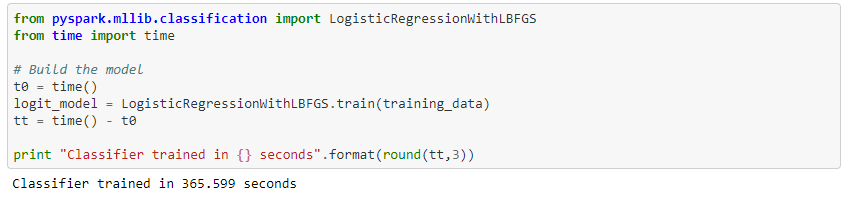


## Queries as DataFrame operations

Spark DataFrame provides a domain-specific language for structured data manipulation. This language includes methods we can concatenate in order to do selection, filtering, grouping, etc. For example, let's say we want to count how many interactions are there for each protocol type. We can proceed as follows.



LOGISTIC REGRESSION:

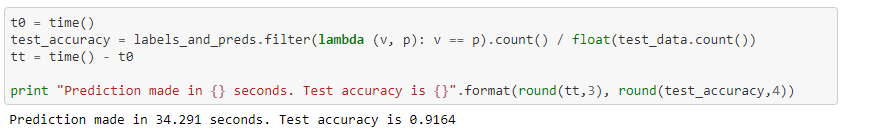


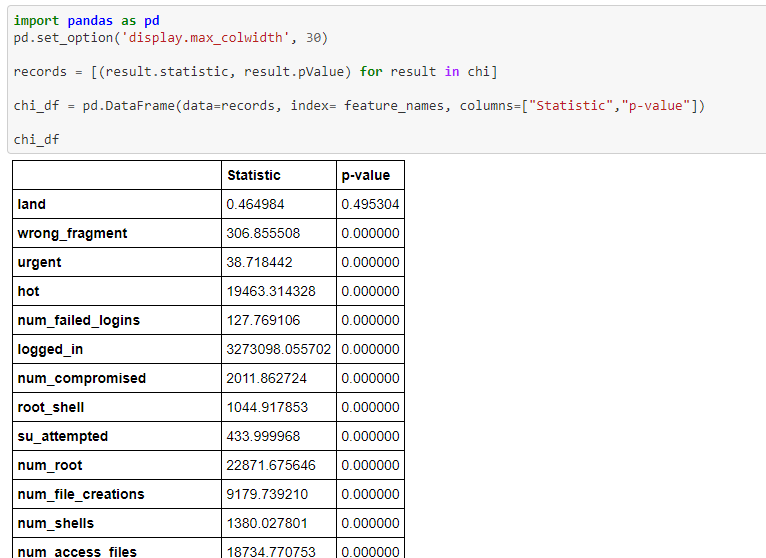
### Evaluating the model on new data

In order to measure the classification error on our test data, we use map on the test\_data RDD and the model to predict each test point class.

Classification results are returned in pars, with the actual test label and the predicted one. This is used to calculate the classification error by using filter and count as follows.







# MLlib: Decision Trees

[Introduction to Spark with Python, by Jose A. Dianes](https://github.com/jadianes/spark-py-notebooks)

In this notebook we will use Spark's machine learning library [MLlib](https://spark.apache.org/docs/latest/mllib-guide.html) to build a **Decision Tree** classifier for network attack detection. We will use the complete [KDD Cup 1999](http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html) datasets in order to test Spark capabilities with large datasets.

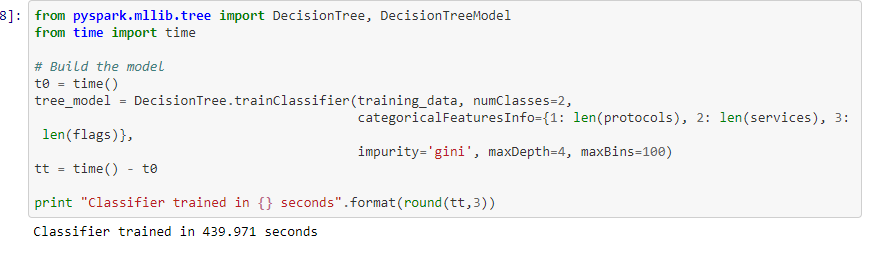
Decision trees are a popular machine learning tool in part because they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and are able to capture non-linearities and feature interactions. In this notebook, we will first train a classification tree including every single predictor. Then we will use our results to perform model selection. Once we find out the most important ones (the main splits in the tree) we will build a minimal tree using just three of them (the first two levels of the tree in order to compare performance and accuracy.

## Detecting network attacks using Decision Trees

In this section we will train a classification tree that, as we did with logistic regression, will predict if a network interaction is either normal or attack.

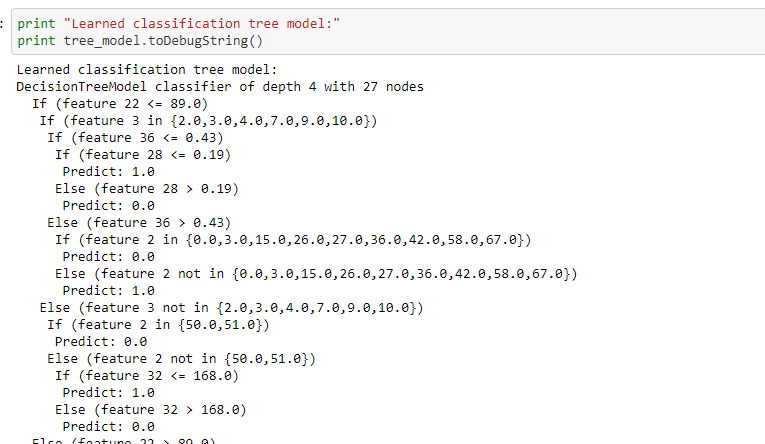
Training a classification tree using [MLlib](https://spark.apache.org/docs/latest/mllib-decision-tree.html) requires some parameters:

* Training data
* Num classes
* Categorical features info: a map from column to categorical variables arity. This is optional, although it should increase model accuracy. However it requires that we know the levels in our categorical variables in advance. second we need to parse our data to convert labels to integer values within the arity range.
* Impurity metric
* Tree maximum depth
* And tree maximum number of bins



### Interpreting the model

Understanding our tree splits is a great exercise in order to explain our classification labels in terms of predictors and the values they take. Using the toDebugString method in our three model we can obtain a lot of information regarding splits, nodes, etc.



Reference :

<https://hub.docker.com/r/jupyter/pyspark-notebook/>