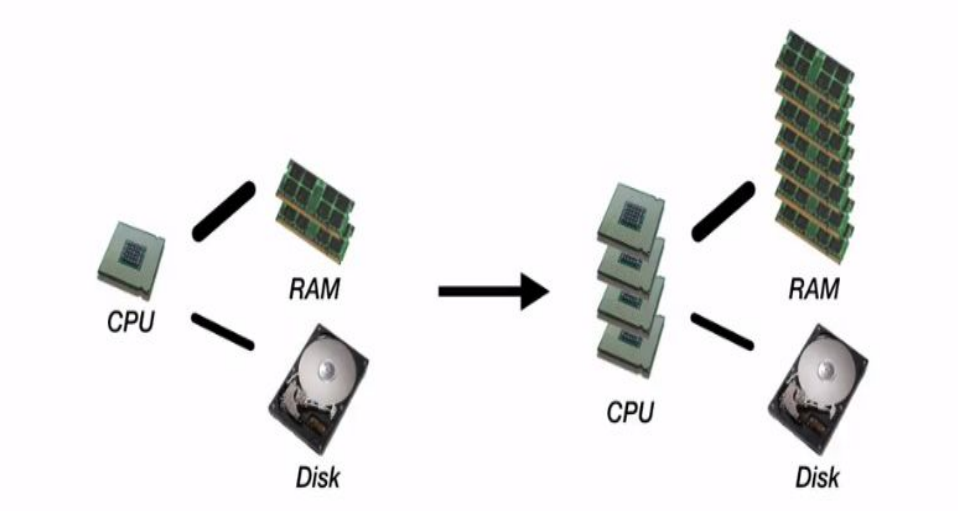
Apache Spark for Practical Machine Learning

Why Distributed Computing ??

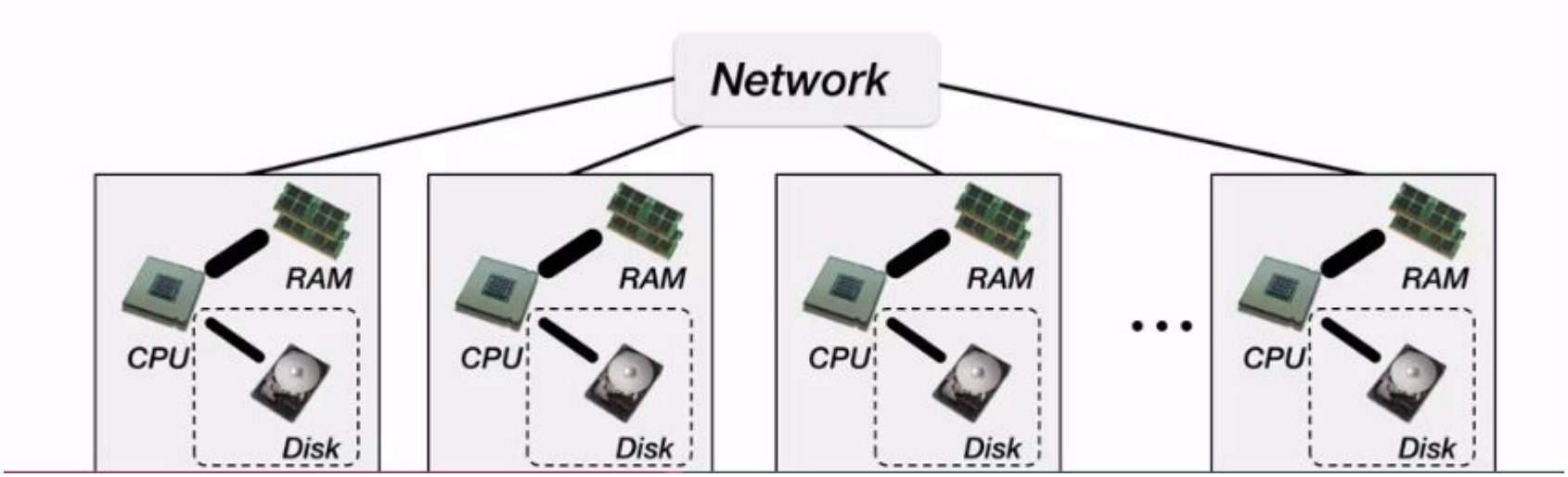
- Why can’t Traditional Tools (Matlab, R, Excel) cannot be used for processing large datasets

  - They typically run on single machine.
  - Need more hardware to store / process data

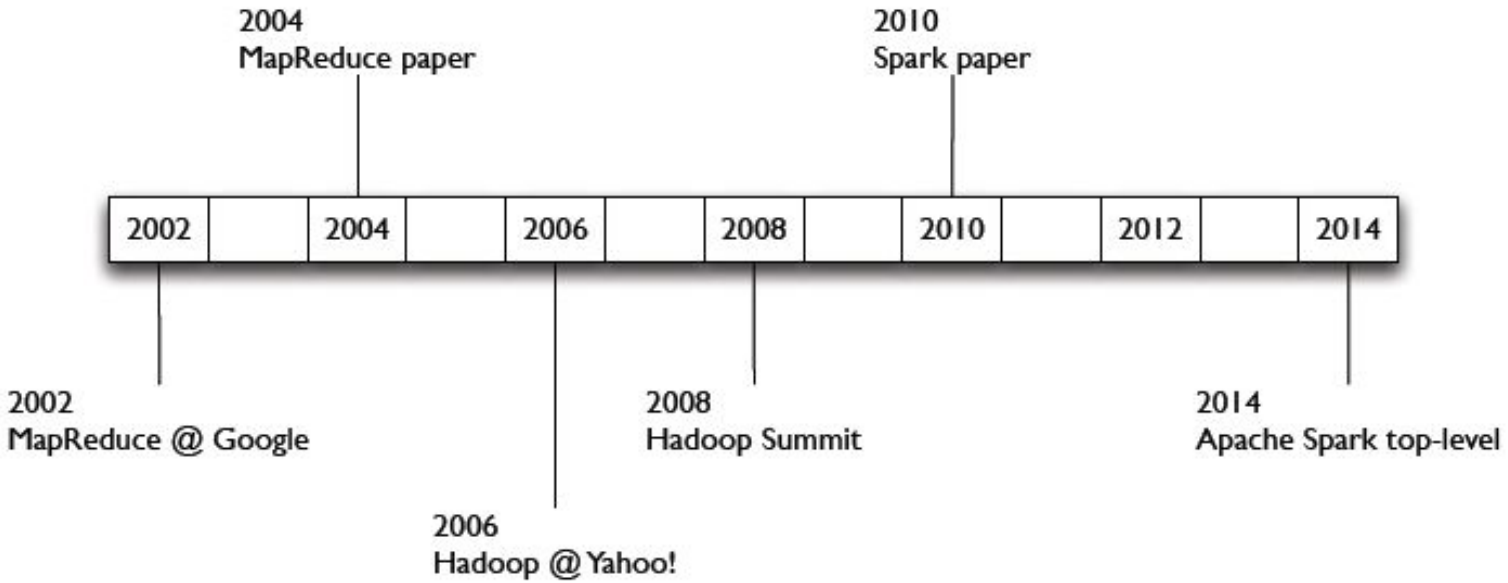
- **Scale – up** the machine (large machine)
  - Good Idea ! Actually it works faster
  - But need Specialized hardware (expensive)
  - Scaling can be done to a certain extent



- **Scale out**
  - Many small machine connected them over network in distributed setting
    - Better alternative , as nodes can easily be added
    - Commodity hardware
    - Network Communication , Software Complexity



Cluster Computing Platforms

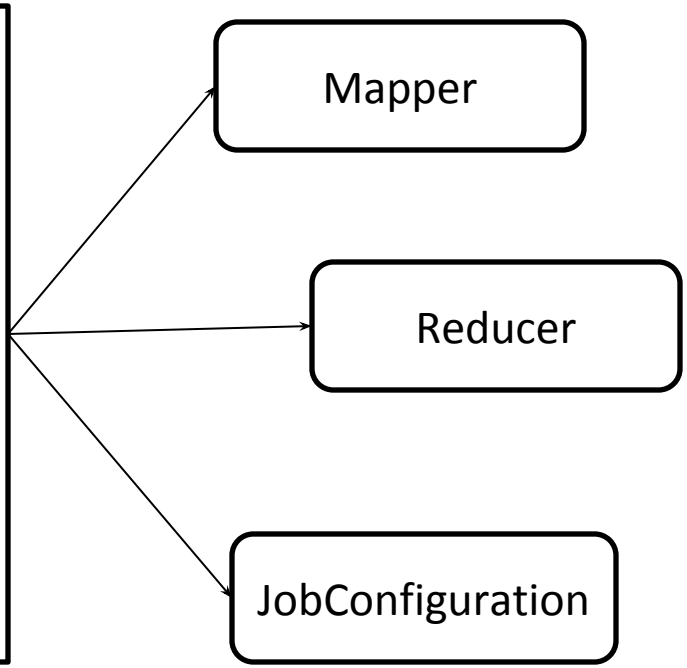


Hadoop vs. Spark

- Two main challenges with Hadoop
  - Configuring Cluster
  - Complex Programming model (MapReduce)

```
void map (String doc_id, String text)
for each word w in segment(text)
emit(w, "1");

void reduce (String word, Iterator
group):
int count = 0;
for each pc in group:
count += Int(pc);
emit(word, String(count));
```

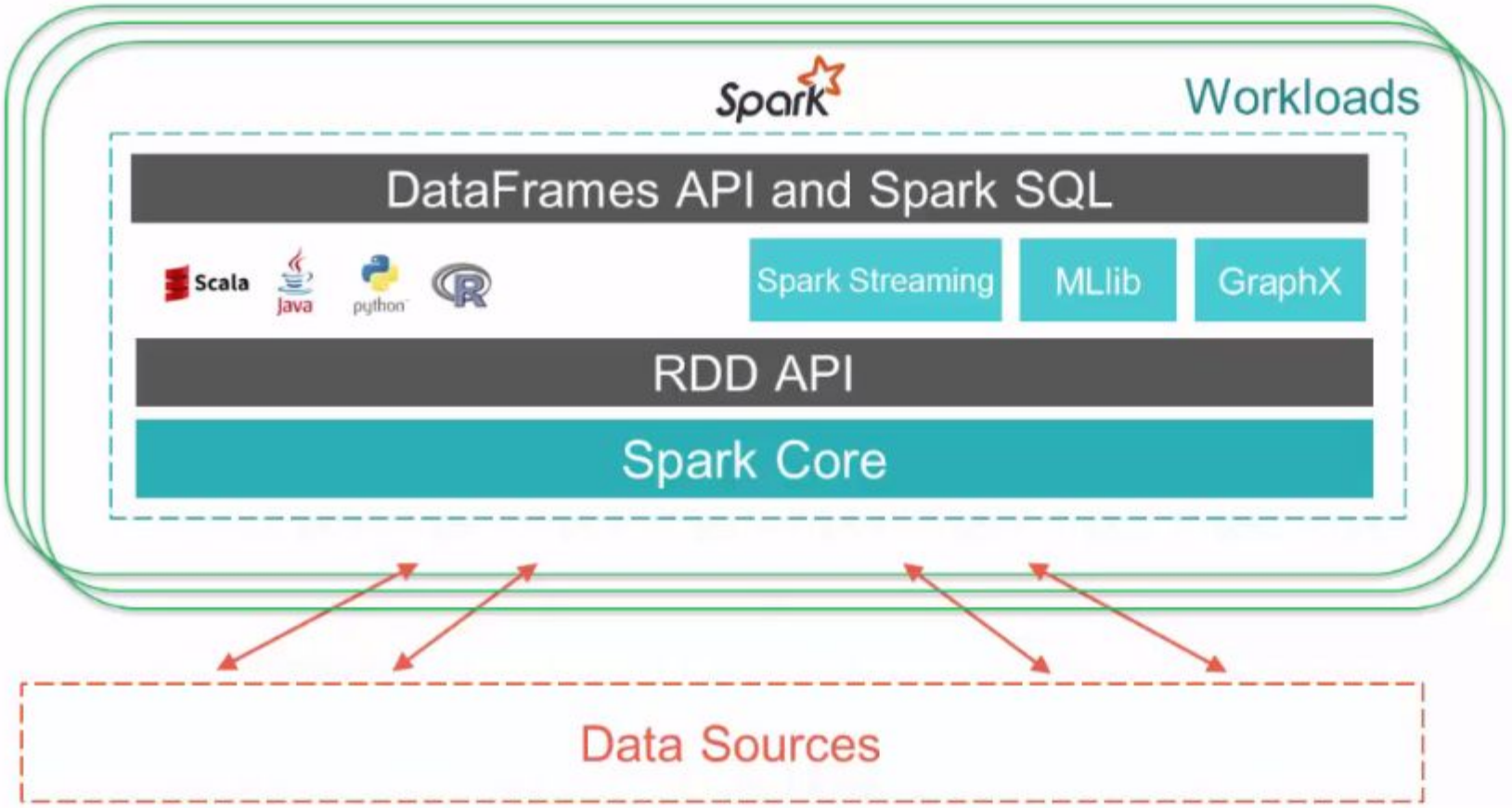


Apache Spark

- Open source cluster computing engine

- Why spark for large scale machine learning?
  - Fast iterative computation
  - Communication primitive
  - Provides API for scala, python and java
  - Interactive shell
  - Many high level libraries are available for building machine learning pipelines

Components of Spark



- Spark core , RDD API – low level access to spark functionality
- MLlib, Streaming, GraphX , DataFrames API and SparkSQL – high level operation (top of RDD API)
- In case of Hadoop, if a new functionality is required, it is added as a tool where as in Spark, it is added as an package.

Resilient distributed Datasets

- **RDD ~ data stored in Spark.**
- Immutable.
- Collection of objects distributed across the cluster.
- In order to manipulate the data in RDD, two operations are typically supported: actions and transformations.

Installation

Download recent version of spark

http://spark.apache.org/downloads.html

Version : 2.0.0

Prebuilt for Hadoop version : 2.6.0

Installation

Untar the Zip file and include it to /usr/local directory

```
mkdir /usr/local/SPARK_INSTALL
cp /Downloads/spark-2.0.0-bin-hadoop2.6.tgz /usr/local/SPARK_INSTALL
cd /usr/local/SPARK_INSTALL
tar xvzf /usr/local/SPARK_INSTALL
mv spark-2.0.0-bin-hadoop2.6.tgz spark
```

Open bashrc

```
nano ~/.bashrc
```

Include the environment variable

```
export $SPARK_HOME=/usr/local/SPARK_INSTALL/spark\
export $PATH=$PATH:$SPARK_HOME\bin
```

Source bashrc

```
source ~/.bashrc
```

Spark Essentials

- Interactive shells

  (In the installation directory of spark)

  - bin/spark-shell (scala)
  - bin/pyspark (python)

Spark Context

- Spark Program ~ Create a Spark Context object.
- Spark Context
  - Access to the cluster.
- In the interactive shell, they are created automatically and available in variable sc.
- Separate programs , then initialize SparkContext in the application.

```
sc
```

```
<pyspark.context.SparkContext object at 0x8a7398c>
```

- Master parameter determines which cluster to use

Master	Description
local	Run spark locally with one worker thread
local [k]	Run spark locally with k worker threads
spark://HOST:PORT	connect to spark standalone cluster PORT depends on config (7077 by default)
mesos://HOST:PORT	connect to spark Mesos cluster PORT depends on config (5050 by default)

Spark Context

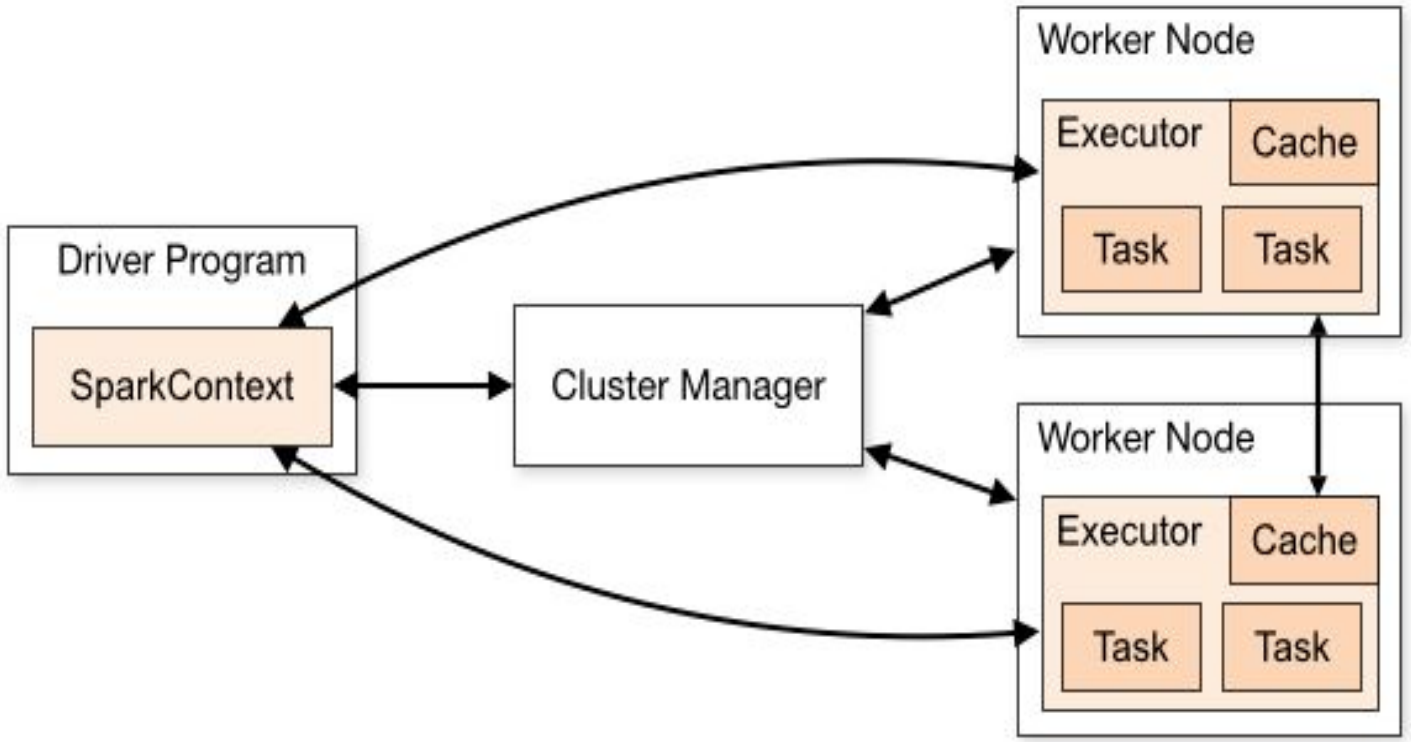
```
import org.apache.spark.SparkContext
import org.apache.spark.SparkConf

from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName(appName)\
        .setMaster(master)

sc = SparkContext(conf=conf)
```

Spark Components



- Driver program – main Program that submits the job to the cluster

  - Connects to cluster manager for resource allocation
  - Acquire executors on cluster nodes – worker process which runs computations and store data
  - Send app code to executors
  - Send tasks for executors to run

RDD Creation

- Calling parallelize method on spark context

  data = [1, 2, 3, 4, 5]

  distData = sc.parallelize(data)

- The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.

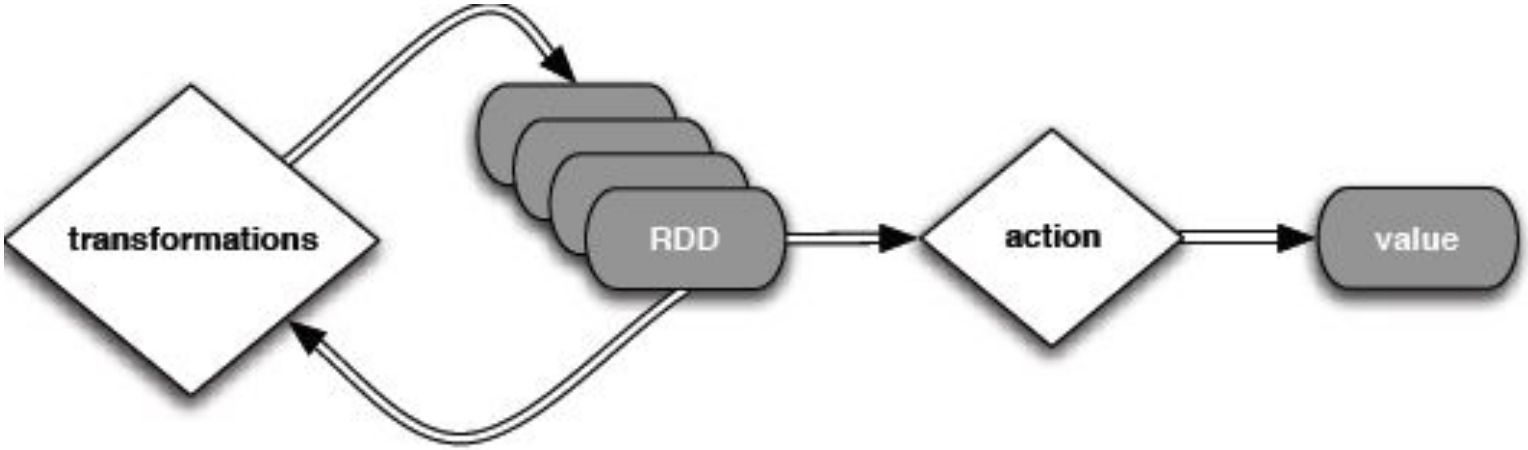
- Create RDD from local text file / HDFS / any external database

  wordRDD=sc.textFile(“/path/to/ReadMe.md”)

- On parallelize, the data is divided into partitions and is stored at the worker nodes.
- Each partition is typically of size 64 MB.
- Smaller partitions can be created by passing an additional parameter to parallelize method

```
sc.parallelize(data,10)
```

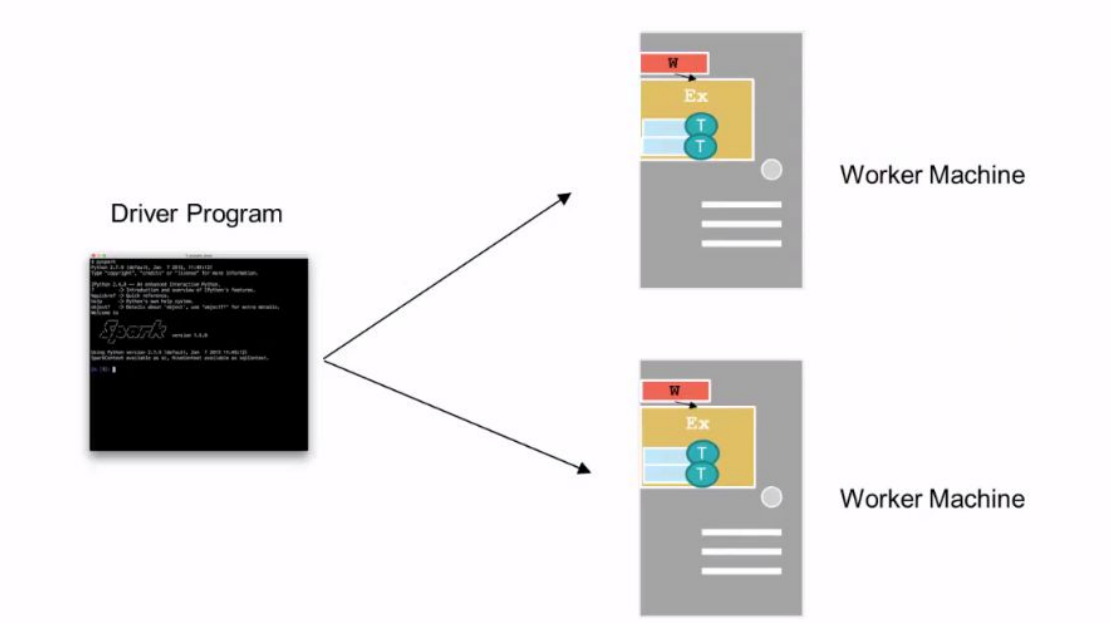
Transformations & Actions



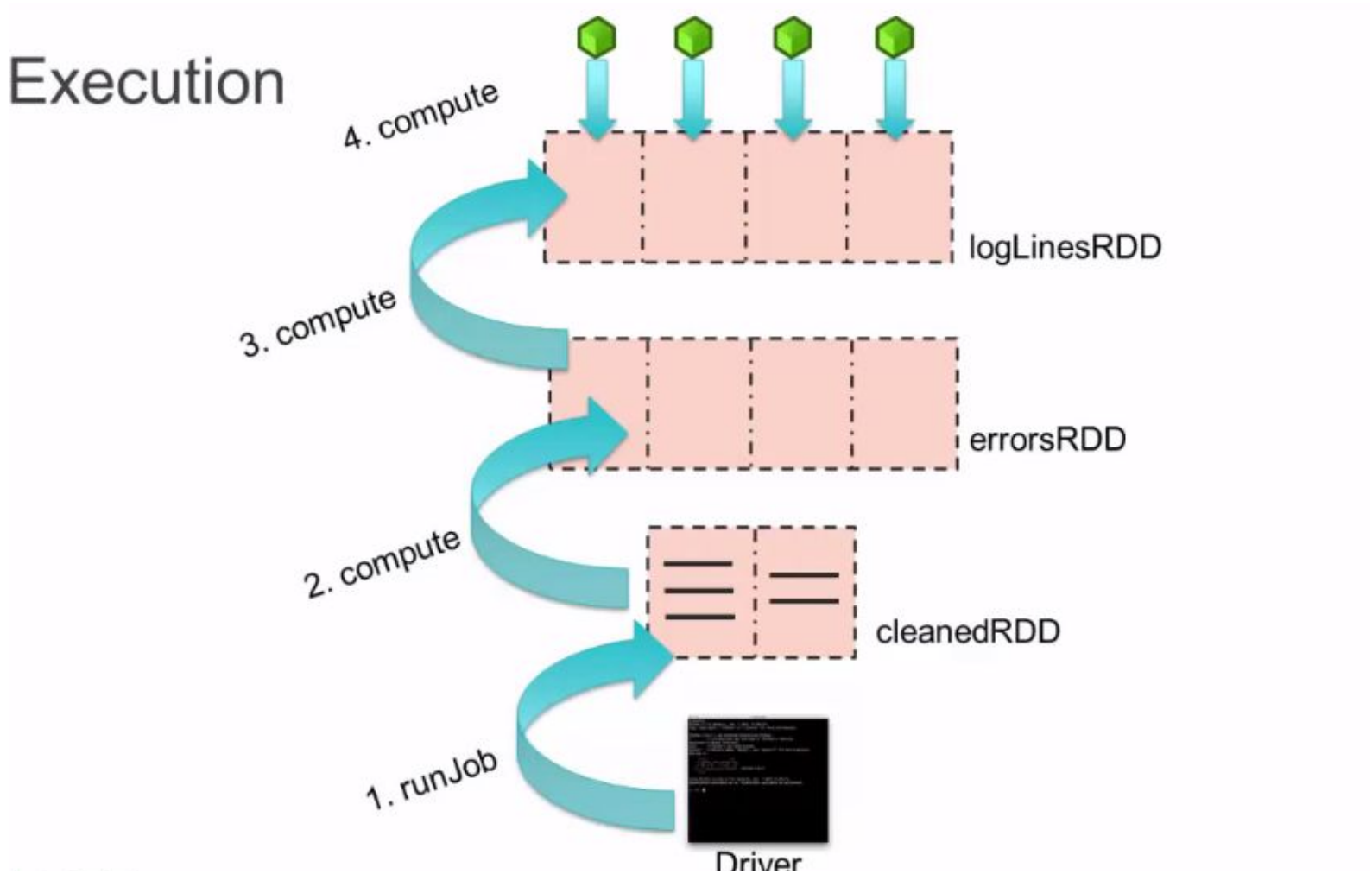
- Transformations are methods which creates new RDD from an existing one.
- Transformations are lazy
  - They are not computed immediately
- Actions computes value from RDD.
- Action causes execution to begin. Launch spark jobs and related transformations are computed.

Typical Workflow in Spark

- Driver creates DAG (Directed Acyclic Graph) for work to be done and sends it to the worker nodes in the cluster . Cluster will return results to driver program



Data Lineage



Usage of Functions

- Lambda expressions, for simple functions that can be written as an expression. (single statement functions)
- Local defs inside the function calling into Spark, for longer code.
- Top-level functions in a module.

```
def f(x):
    return x*2

g = lambda x : x*2
```

```
        def myFunc(s):
              words = s.split(" ")
              return len(words)

conf = SparkConf().setAppName(‘sample’)\
          .setMaster(‘local’)

  sc = SparkContext(conf=conf)
  sc.textFile("file.txt").map(myFunc)
```

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

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)

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

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

Closures

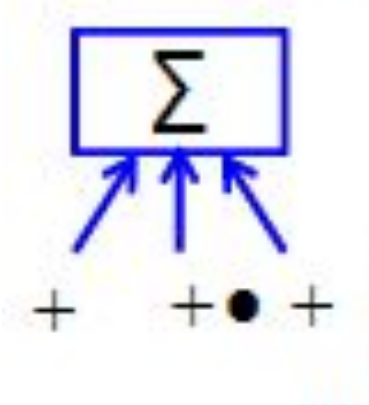
- The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD
- Driver -> (tasks + closure(serialized format)) -> Executors
- Executors cannot send the value back to driver

Closures

- Define the scope and life cycle of variables and methods when executing code across a cluster.

```
data= [1,2,3,4]
counter = 0
rdd = sc.parallelize(data)
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)
print("Counter value: ", counter)
```

- Spark supports two types of shared variables:
- *broadcast variables*, which can be used to cache a value in memory on all nodes.
- *accumulators*, which are variables that are only “added” to, such as counters and sums.



Accumulators

- Aggregate values from workers back to driver
- Only driver can access the value of accumulator
- for tasks, accumulators are write only
- used to count errors seen in RDD across worker nodes

Creating an accumulator with an initial value of 0

```
accum = sc.accumulator(0)
```

Accumulator can be manipulated either using += or add function

```
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
```

Driver program can access the value

```
accum.value
```

Broadcast Variables

- Efficiently send large read – only values to all workers.
- Saved at workers for use in one or more operations.
- Sending read only lookup table to all nodes

Creating a broadcast variable

```
broadcastVar = sc.broadcast([1, 2, 3])
```

Accessing the broadcast variable

```
broadcastVar.value
```

```
[1, 2, 3]
```

User Interface

- Each driver program has a web UI, typically running on port 4040, that displays information about running tasks, executors, and storage usage.

- Simply go to http://localhost:4040 in a web browser to access this UI.

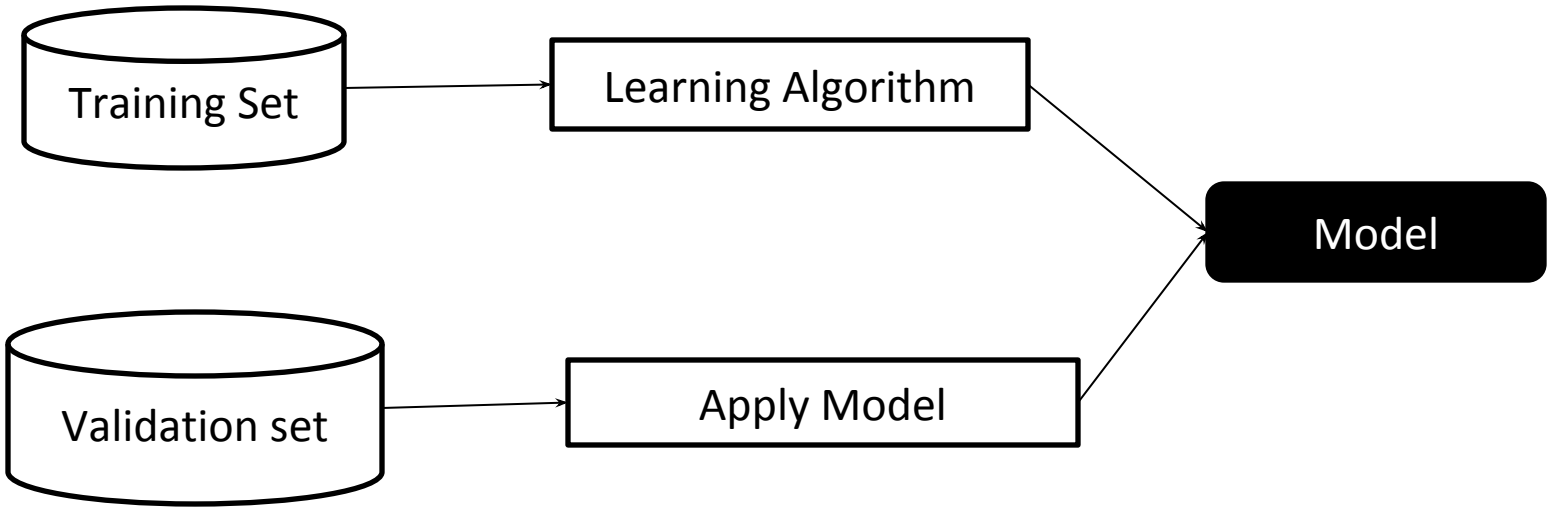
MLlib

- Machine learning package available in Spark.
- It is shipped with Spark 0.8.
- Started as a project in UC Berkeley AMPLab.
- It consists of common learning algorithms and utilities including. classification, regression, clustering, collaborative filtering, dimensionality reduction.

MLlib

- Machine learning has to be easy and scalable
  - Capable of learning from large datasets.
  - Easy way to build machine learning applications.

Classifier - Phases

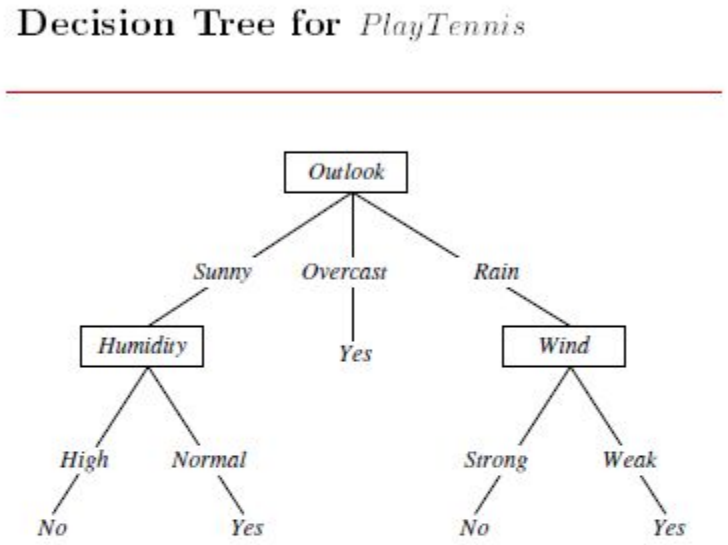


Random Forest Classifier

- Ensemble of decision trees
- Decision Trees
  - Simple means of inducing rules
    - if (Age is x) and (income is y) then sanction loan

Sample Decision Tree

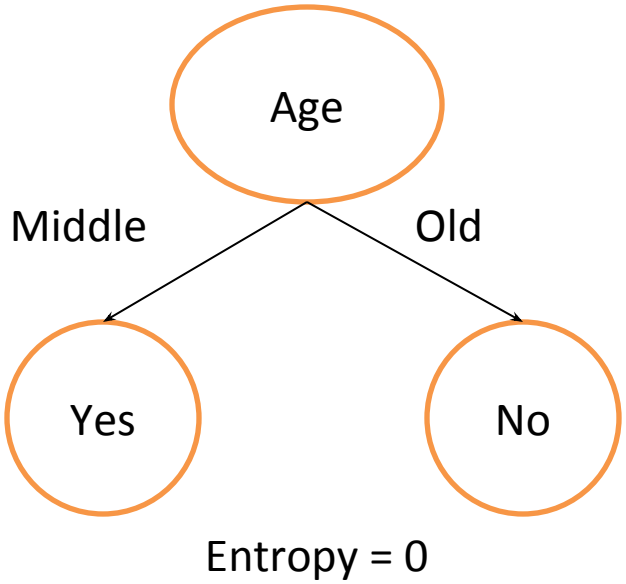
Day	Outlook	Temperature	Humidity	Wind	PlayTenni
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No



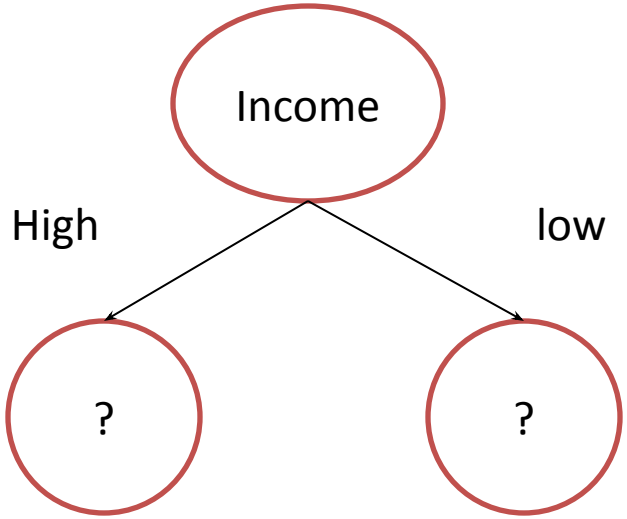
How attributes are selected ?

- Using Metrics
  - Information Gain
  - Entropy

Age	Income	Label
Middle	High	Yes
Middle	Low	Yes
Old	High	No
Old	Low	No



Entropy = 0



Entropy = 1

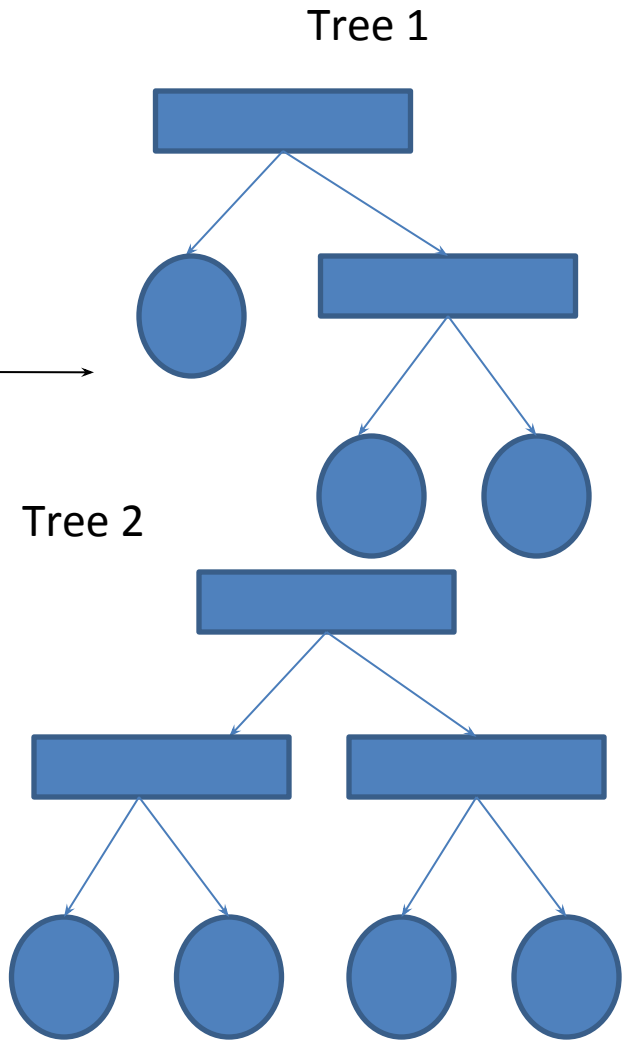
Random Forest classifier

- Problem with decision tree
  - Overfits the training data

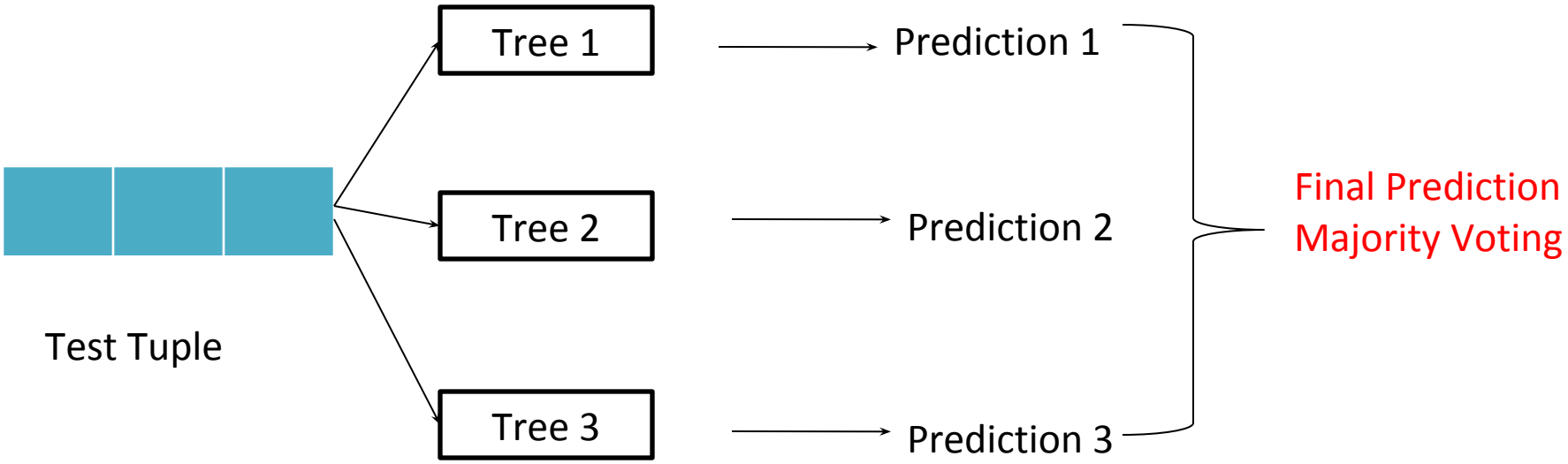
Day	Outlook	Temperature	Humidity	Wind	PlayTenni
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Outlook	Temperature	Play_Tennis
Tuple1		
Tuple 4		
Tuple 8		

Temperature	Wind	Play_Tennis
Tuple 2		
Tuple 5		
Tuple 8		



Train & Test Model



Data

- Mllib supports different datatypes (local)
  - Vectors (columnar values)
    - Sparse
    - Dense
  - LabeledPoint
    - associate label with a vector
    - label field ---- > double value
    - features field ---- > Vector

```
from pyspark.mllib.regression import LabeledPoint
pos = LabeledPoint (1.0, [1.0,0.0,3.0])
```

MLlib Package- Radom Forest Classifier

- Random Forest Classifier
  - pyspark.mllib.tree
  - import RandomForest, RandomForestModel
- Train
  - model = RandomForest.trainClassifier (Data,
    numClasses =2,
    CategoricalFeatureInfo{ Map(0->2,4>10)},
    impurity="gini",
    maxDepth=4,
    maxBins=32,
    numTrees=3)

- Problem Specification Parameters
  - Algorithm
  - numClasses
  - categoricalFeaturesInfo
- Stopping Criteria
  - maxDepth
  - minInstancesPerNode
  - minInfoGain
- Tunable Parameters
  - Impurity

Testing Model using MLlib

- testdata – dataset with many tuples
- predictions=model.predict(testdata.map(lambda x:x.features)

- labelsAndPredictions=testdata\
  .map(lambda lp:lp.label)\
  .zip(predictions)

Evaluation

- testErr=labelsAndPredictions.
  .filter(lambda (v,p) : v!=p).count()/float(testData.count())

- print(‘Test Error = ‘ + str(testErr))

- model.save(sc,”MyModelPath”)
- sameModel = RandomForestModel.load(sc,”MyModelPath”)

MLlib

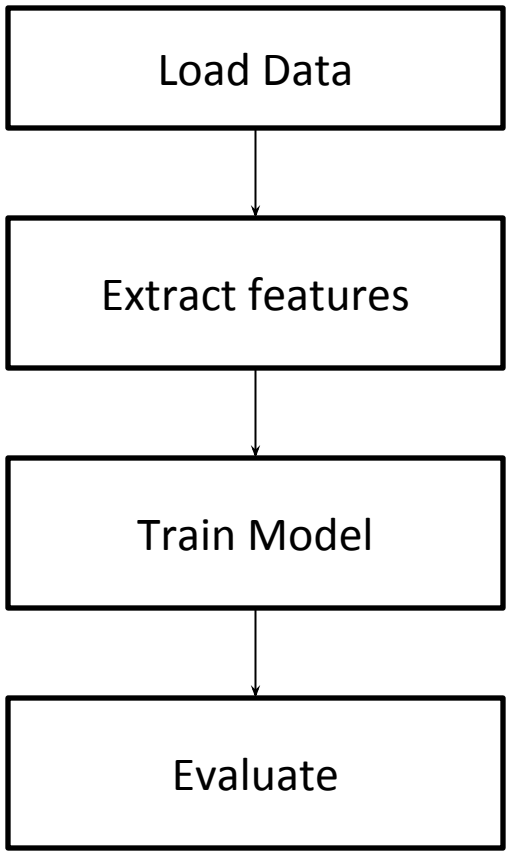
- Machine learning has to be easy and scalable
  - Capable of learning from large datasets.
  - Easy way to build machine learning applications.

Machine Learning Workflow

- Scalable -- Expandable (it should work even if the data grows enormously)
- Machine learning pipeline components
  - Feature Extraction
  - Supervised Learning
  - Model Evaluation
  - Exploratory data analysis

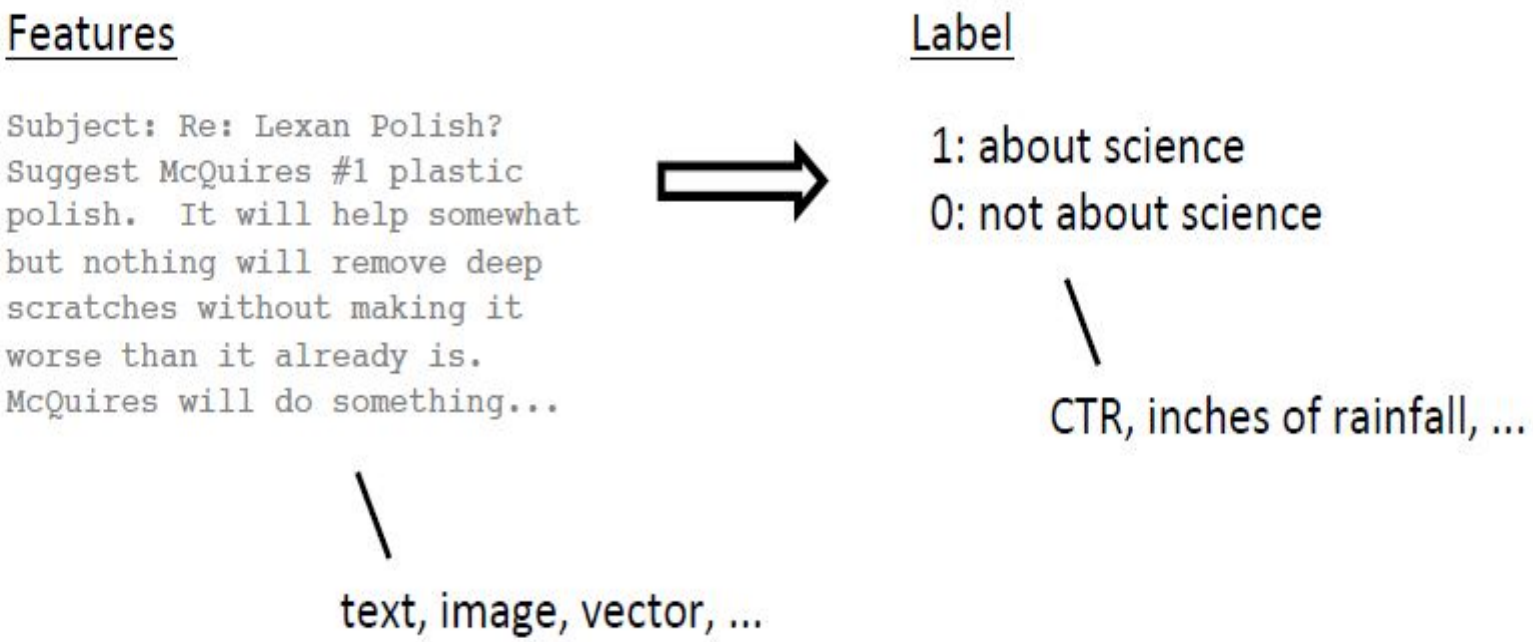
ML workflow

- Typical ML workflow



Text Classification

- Given text predict its topic



Training & Testing

Training

Given labeled data:

RDD of (features, label)

```
Subject: Re: Lexan Polish?
Suggest McQuires #1 plastic
polish.  It will help...
```
Label 0

```
Subject: RIPEM FAQ
RIPEM is a program which
performs Privacy Enhanced...
```
Label 1

...

Learn a model.

Testing/Production

Given new unlabeled data:

RDD of features

```
Subject: Apollo Training
The Apollo astronauts also
trained at (in) Meteor...
```

```
Subject: A demo of Nonsense
How can you lie about
something that no one...
```

Use model to make predictions.

Main Challenges

- RDD representation
  - RDD is immutable
  - Adding new fields is not possible
  - Creation of more RDDs
- Process of feature extraction is complex
  - iterative process
  - Workflow is not generally shared
  - Creates problem during Production environment
- ML Tuning
  - Best Model
  - Parameter Tuning
  - Validations

ML Pipeline Concepts

- DataFrame (RDD representation)
- Transformer
  - Feature Extractors
  - Classifiers
- Estimator
  - Models
- Pipeline
  - Represents the workflow

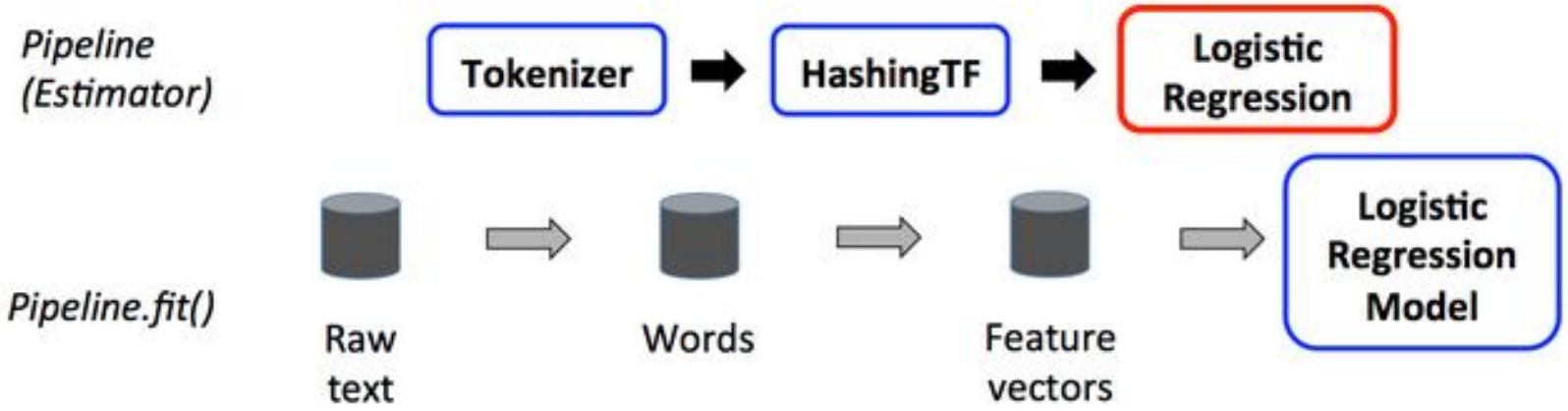
Text Processing

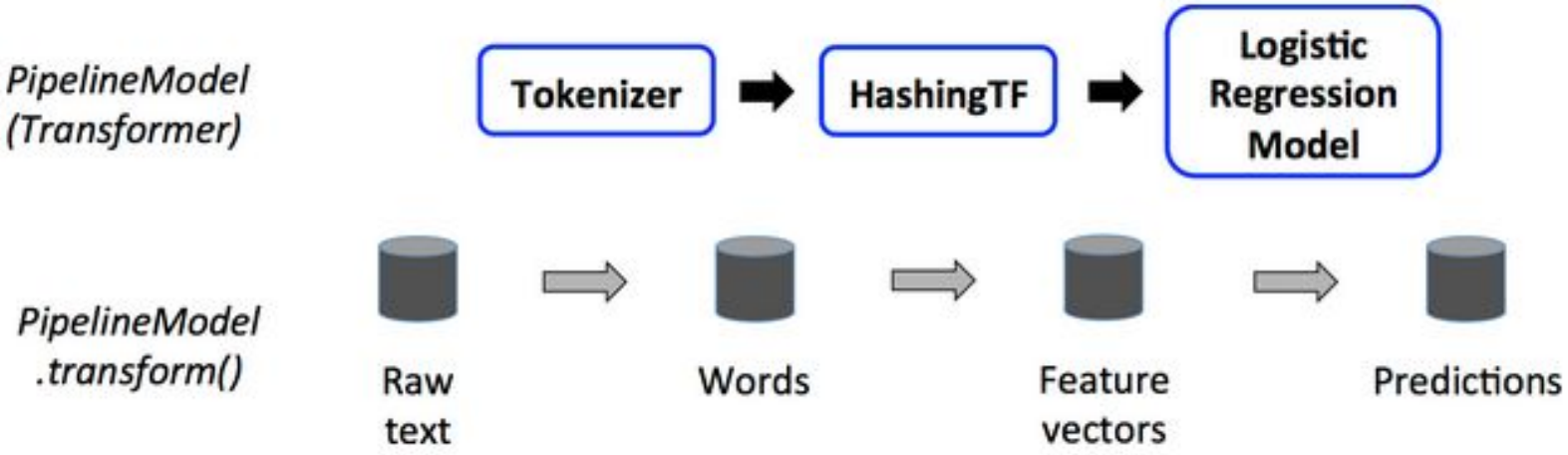
- Split each document’s text into words.
- Convert each document’s words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

ML workflow is pipeline which contains sequence of pipeline stages

Transformer and Estimator

- Each Pipeline Stage is either transformer or estimator
- Transformer typically accepts a data frame and returns a new dataframe with added columns ~ transform()
- Estimator accepts a data frame and provides a transformer ~ fit()





Identical feature processing steps for both training and test data

DataFrame

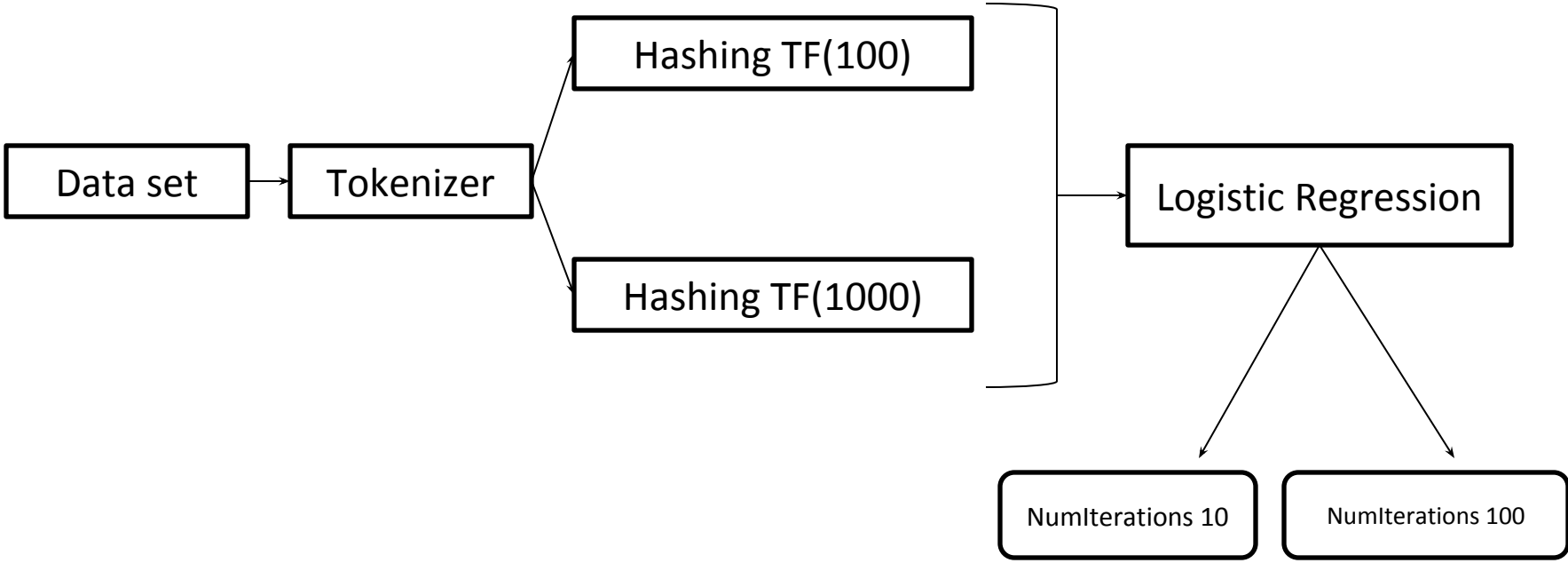
- df = spark.createDataFrame([(0, "a b c d e spark", 1.0),
(1, "b d", 0.0),
(2, "spark f g h", 1.0),
(3, "hadoop mapreduce", 0.0)] , ["id", "text", "label"])

  tokenizer = Tokenizer(inputCol="text", outputCol="words")
  hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
  lr = LogisticRegression(maxIter=10)
  pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

model = pipeline.fir(df) # training
selected = model.transform(test) # add a new column prediction

ML Tuning

- Find the best models (or) parameters for given task.
- Tuning
  - Featurization
  - Estimator



ML Evaluators

- Cross Validator / TrainValidationSplit
- Requires
  - Estimator (pipeline)
  - Parameter Grid
  - Evaluator

```
paramGrid = ParamGridBuilder() \
    .addGrid(hashingTF.numFeatures, [10, 100, 1000]) \
    .addGrid(lr.regParam, [0.1, 0.01]) \ .build()

crossval = CrossValidator(estimator=pipeline,
    estimatorParamMaps=paramGrid,
    evaluator=BinaryClassificationEvaluator(), numFolds=2)
```

References

Books :

- **Learning Spark: Lightning-Fast Big Data Analysis by** Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia

Online Courses :

edx course - **CS120x Distributed Machine Learning with Apache Spark**

Web Resources:

http://spark.apache.org/docs/latest/programming-guide.html

http://spark.apache.org/docs/latest/quick-start.html