Spark Overview
Overview

● What is Spark?

● How Spark works?
  ○ Mechanism
  ○ Logistic Regression Model

● Why Spark?

● How to leverage Spark?
What is Spark

- **Hadoop/YARN:**
  - strong in processing large files parallelly
  - synchronization barrier when persisting data to the disk.
  - MapReduce: launch mapper & reducer, R/W to disk, back to queue and get resource

- **Spark:**
  - in-memory processing
  - iterative and interactive data analysis
  - compare to MapReduce, supports more complex and interactive applications
Hadoop MapReduce

- Slow due to replication, serialization, and disk IO
- Inefficient for:
  - Iterative algorithms (Machine Learning, Graphs & Network Analysis)
  - Interactive Data Mining (R, Excel, Searching)
Spark In-memory Processing

Iterative:
1. Extract a working set
2. Cache it
3. Query it repeatedly

Interactive:
1. Extract a working set
2. Cache it
3. Query it repeatedly
Spark Ecosystem

- Shark SQL
- Spark Streaming
- GraphX
- MLLib

Spark

HDFS / Hadoop Storage

Mesos/YARN Resource Manager
Overview

● What is Spark?

● How Spark works?
  ○ Mechanism
  ○ Logistic Regression Model

● Why Spark?

● How to leverage Spark?
How Spark Works - SparkContext

Driver Program

```java
sc = new SparkContext
rDD = sc.textfile("hdfs://...")
rDD.filter(...)
rDD.Cache
rDD.Count
rDD.map
```

Writes

User (Developer)
How Spark Works - RDD

Driver Program

```
sc=new SparkContext
rDD=sc.textfile("hdfs://...")
rDD.filter(...)
rDD.Cache
rDD.Count
rDD.map
```

Writes

User (Developer)

- Partitions of Data
- Dependencies between partitions

RDD (Resilient Distributed Dataset)

Storage Types:
- MEMORY_ONLY
- MEMORY_AND_DISK
- DISK_ONLY
...

- Fault Tolerant
- Controlled partitioning to optimize data placement
- manipulated using rich set of operators.
How Spark Works - RDD operations

**Transformations**
- Create a new dataset from an existing one.
- Lazy in nature, executed only when some action is performed.
- Example
  - Map(func)
  - Filter(func)
  - Distinct()

**Actions**
- Returns a value or exports data after performing a computation.
- Example:
  - Count()
  - Reduce(func)
  - Collect
  - Take()

**Persistence**
- Caching dataset in-memory for future operations
- Store on disk or RAM or mixed
- Example:
  - Persist()
  - Cache()
How Spark Works: Word Count

```python
sparkContext.textFiles("hdfs://...")
```

RDD[String]
How Spark Works: Word Count

```scala
sparkContext.textFiles("hdfs://...")
  .map(line => line.split("\s"))
```

RDD[String]
RDD[List[String]]
How Spark Works: Word Count

```scala
sparkContext.textFiles("hdfs://...")
  .map(line => line.split("\\s"))
  .map(word => (word, 1))
```

- RDD[String]
- RDD[List[String]]
- RDD[(String, Int)]
How Spark Works: Word Count

```
sparkContext.textFiles("hdfs://...")
  .map(line => line.split("\s"))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)
```
How Spark Works: Word Count

```
sparkContext.textFiles("hdfs://...")
  .map(line => line.split("\s"))
  .map(word => (word, 1))
  .reduceByKey((a, b) => a + b)
  .collect()
```

RDD[String]
RDD[List[String]]
RDD[(String, Int)]
RDD[(String, Int)]
Array[(String, Int)]
How Spark Works - Actions

- Parallel Operations
  - map
  - reduce
# How Spark Works - Actions

- **Parallel Operations**

  - `map`
  - `filter`
  - `groupBy`
  - `sort`
  - `union`
  - `join`
  - `leftOuterJoin`
  - `rightOuterJoin`
  - `reduce`
  - `count`
  - `fold`
  - `reduceByKey`
  - `groupByKey`
  - `cogroup`
  - `cross`
  - `zip`
  - `sample`
  - `take`
  - `first`
  - `partitionBy`
  - `mapWith`
  - `pipe`
  - `save`
How Spark Works - Stages

DAG (Directed Acyclic Graph).

Each stage is executed as a series of Task (one Task for each Partition).

Stage 1

Stage 2
Spark Programming - Tasks

Task is the fundamental unit of execution in Spark

HDFS / RDD
Fetch Input
Execute Task
Write Output
HDFS / RDD / intermediate shuffle output

Core 1
Fetch Input
Execute Task
Write Output

Core 2
Fetch Input
Execute Task
Write Output

Core 3
Fetch Input
Execute Task
Write Output
How Spark Works - Summary

- SparkContext
- Resilient Distributed Datasets (RDDs)
- Parallel Operations

- Shared Variables
  - Broadcast Variables - read-only
  - Accumulators
## Compare Hadoop and Spark

<table>
<thead>
<tr>
<th></th>
<th>Traditional OS</th>
<th>Hadoop</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Storage</strong></td>
<td>File System</td>
<td>HDFS</td>
<td>HDFS</td>
</tr>
<tr>
<td><strong>Schedule</strong></td>
<td>Processes</td>
<td>MapReduce</td>
<td>Computation Graph</td>
</tr>
<tr>
<td><strong>I/O</strong></td>
<td>Disk</td>
<td></td>
<td>Cache(in memory) and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>shared data</td>
</tr>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Duplication and Disk I/O</td>
<td>Hash partition and auto-reconstruction</td>
<td></td>
</tr>
</tbody>
</table>
Overview

● What is Spark?

● How Spark works?
  ○ Mechanism
  ○ Logistic Regression Model

● Why Spark?

● How to leverage Spark?
Spark - LogisticRegressionModel

1. Initialize spark JavaSparkContext

2. Prepare data set

3. Train LR model

4. Evaluation
1. Initializing Spark

1. JavaSparkContext: tell Spark how to access to the cluster
2. SparkConf: setting - a hashmap of <String,String>
   a. required: AppName, Master, more default configuration

```java
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc = new JavaSparkContext(conf);
```
2. Prepare Dataset

1. From Parallelized Collections

```java
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

2. From External DataSets

```java
JavaRDD<String> distFile = sc.textFile("data.txt"); OR
JavaRDD<String> distFile = sc.textFile("hdfs://data.txt");
```

3. Passing Functions to Spark

```java
class ParseLabeledPoint implements Function<String, LabeledPoint> {
    public LabeledPoint call(String s) {
        for (int i = 0; i < len; i++) {
            x[i] = Double.parseDouble(tokens[i]);
        }
        return new LabeledPoint(y, Vectors.dense(x));
    }
}
```
3. Train LogisticRegressionModel

Train the model

```scala
/*
 * @param input RDD of (label, array of features) pairs.
 * @param numIterations Number of iterations of gradient descent to run.
 * @param stepSize Step size to be used for each iteration of gradient descent.
 * @param miniBatchFraction Fraction of data to be used per iteration.
 */
LogisticRegressionModel lrModel = LogisticRegressionWithSGD.train(data, iterations, stepSize, miniBatchFraction);
```
4. Calculate Score - Evaluation

1. Convert LogisticRegressionModel to PMML model

   ```java
   pmmlModel = new PMMLSparkLogisticRegressionModel()
   .adaptMLModelToPMML(lrModel, partialPmmlModel);
   ```

2. Prepare DataSet and calculate score

   ```java
   //use LogisticRegressionModel
   JavaRDD<Vector> evalVectors = lines.map(new ParseVector());
   List<Double> evalList = lrModel.predict(evalVectors).collect();
   //use PMMLEvaluator
   RegressionModelEvaluator evaluator = new RegressionModelEvaluator(pmml);    
   List<Double> evalResult = evaluator.evaluate(evalData);
   //compare two evaluator results
   for (...) {
       Assert.assertEqual(getPMMLEvaluatorResult(i), sparkEvalList.get(i), DELTA);
   }
   ```
Overview

● What is Spark?

● How Spark works?
  ○ Mechanism
  ○ Logistic Regression Model

● Why Spark?

● How to leverage Spark?
Why Spark? - scalability & performance

1. leverage the memory of the cluster for in-memory processing
2. Computation Graph optimization for parallel execution

Shark: Spark SQL, Hive in Spark
Hive: manage large dataset in distributed storage

Real-world Performance

Figure 2: Logistic regression performance in Hadoop and Spark.
Why Spark? - compatibility

1. compatible with HDFS, HBase, and any Hadoop storage system
Why Spark? - Ease of Use API

1. Expressive API in Java, Scala, and Python
2. Supports more parallel operations

```python
lines = sc.textFile(...) 
lines.filter(lambda s: “ERROR” in s).count()
```

```scala
val lines = sc.textFile(...) 
lines.filter(s => s.contains(“ERROR”)).count()
```

```java
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
    Boolean call(String s) {
        return s.contains(“error”);
    }
}).count();
```
public static class WordCountMapClass extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}

public static class WordCountReduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

Expressive API - MapReduce
public static class WordCountMapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}

public static class WorkdCountReduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

Scala:
val file = spark.textFile("hdfs://...")
val counts = file.map(line => line.split(" "))
            .map(word => (word, 1))
            .reduceByKey(_ + _)

val file = spark.textFile("hdfs://...")
output.collect(word, one);
public static class WordCountMapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
        }
    }
}

public static class WorkdCountReduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

Java 6, Java 7:

JavaRDD<String> file = spark.textFile("hdfs://...");
JavaRDD<String> words = file.map(new FlatMapFunction<String, String>() {
    public Iterable<String> call(String s) { return Arrays.asList(s.split(" ")); }
});
JavaPairRDD<String, Integer> pairs = words.map(new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String s) { return new Tuple2<>(s, 1); }
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
output.collect(key, new IntWritable(sum));
Why Spark? - Third Party Softwares

- **Mahout**
  - Say goodbye to MapReduce
  - Support for Apache Spark
    - Mahout-Spark Shell: facilitate the Mahout data structures, such as Matrix, etc.
  - Support for h2o being explored
  - Support for Apache Flink possibly in future

- **H2o**
  - Sparkling water - embrace in-memory processing with ML algorithm

---

<table>
<thead>
<tr>
<th></th>
<th>Purpose</th>
<th>Language</th>
<th>Storage</th>
<th>Stakeholder</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2o</td>
<td>In-memory ML predictive analysis</td>
<td>Java/R</td>
<td>K/V store</td>
<td>data analyst</td>
</tr>
<tr>
<td>Spark</td>
<td>in-memory processing engine</td>
<td>Scala, support Java/Python</td>
<td>RDD</td>
<td>HDFS user</td>
</tr>
</tbody>
</table>
Why Spark - Third Party Software

- Pig on Spark - Spork
- Other commercial softwares
Overview

● What is Spark?

● How Spark works?
  ○ Mechanism
  ○ Logistic Regression Model

● Why Spark?

● How to leverage Spark?
How to use Spark in Shifu?

1. **train:** LogisticRegressionTrainer
2. **stats & normalize**
3. **eval:** add more evaluation metrics
   - a. precision, recall, F-measure, precision-recall curve
      - pr(), precisionByThreshold(), recallByThreshold().. 
   - b. area under the curves (AUC) - areaUnderPR() 
   - c. receiver operating characteristic (ROC) - areaUnderROC(), roc()
Related Projects

1. Bulk Synchronous Parallel
   a. parallel computing on message-passing
   b. BSP: local computation, global communication, barrier synchronization
   c. graph processing: Pregel, Giraph
   d. scientific computing: Hama
   e. optimize operation DAG: Flink
Take Away - Big Data has moved in-memory

1. In-memory big data has come of age.
2. Spark leverages the cluster memory for iterative and interactive operations
3. Spark is compatible with HDFS, HBase, and any Hadoop storage system
4. Spark powers a stack of high-level tools including Spark SQL, MLlib for machine learning, GraphX, and Spark Streaming
5. Spark has expressive API
Questions
3. Train LogisticRegressionModel (cont.)

2. Calculate weights

```scala
val weightsWithIntercept = optimizer.optimize(data, initialWeightsWithIntercept)
val weights =
    if (addIntercept) { ...
    } else { weightsWithIntercept }
```

3. Gradient Descent optimize()

```
Algorithm 1: GradientDecent Optimize Function

Input: data, gradient, updater, stepSize, numIterations, regParam, miniBatchFraction
Output: weights: Vector
for i < numIterations do
    dataSet ← dataSample(data, miniBatchFraction)
    for data ∈ dataSet do
        (gradientSum, lossSum) ← computeGradient(gradient)
        sumUpSubGradients();
        (weights, regVal) ← computeUpdate(updater, stepSize)
```

4. Training Error - not accessible from LogisticRegressionModel

```scala
logInfo("Last 10 stochastic losses %s".format(stochasticLoss.takeRight(10)))
```

14/07/09 14:10:40 INFO optimization.GradientDescent: Last 10 stochastic losses 0.6931471805599468, 0.5255572298404575,.., 0.3444544005102222, 0.3355921369255156