Distributed Machine Learning
in Hadoop Ecosystem

Pelle Jakovits

14 December 2018, Tartu
Outline

• Distributed Machine learning libraries
  – Apache Mahout, H2O, Spark ML
• Machine learning with Spark ML
• Distributed R with SparkR
• The Big Picture
• Summary
• Exam information
Extensions to distributed computing engines

• Hadoop MapReduce and Apache Spark provide powerful **core engines** for distributed computing applications
  – Fault tolerance, inherent parallelism, high scalability

• We have looked at their extensions for:
  1. Higher level distributed application development
  2. Extensions for specific type of data (Graph, Streaming)

• There are also extensions that provide distributed computing libraries for specific application fields
  – Such as Machine Learning
Distributed Machine Learning libraries

• Exploit distributed computing engines to provide fault tolerant and scalable implementations for most commonly used Machine Learning (ML) methods
  – Clustering, Classification, Feature manipulation, Recommendation systems, etc.

• ML libraries in the Hadoop/Spark ecosystem
  – Apache Mahout
  – H2O
  – Spark ML library
Apache Mahout

• Initially was designed as a ML extension to Hadoop MapReduce
  – Iterative ML algorithms highlighted issues with MapReduce’s dependency on HDFS
• Currently supported backends are:
  – Apache Spark, H2O, Apache Flink
  – Support for Hadoop MapReduce has been deprecated
  – Apache Spark is recommended
• Provides a set of typical machine learning methods
  – Contains both distributed and single machine algorithms
• Provides interfaces for developing new distributed machine learning algorithms:
  – Distributed linear algebra primitives
  – R-Like Scala Domain Specific Language (DSL)
• Not all previously available MapReduce implementations have been ported to new interfaces
ML algorithms

- **Distributed Linear Algebra algorithms**
  - QR decomposition for Matrix factorization
  - Stochastic Principal component analysis (S-PCA) - dimensionality reduction
  - Stochastic Singular Value Decomposition (S-SVD) - dimensionality reduction

- **Clustering**
  - Canopy Clustering *(MR only: k-Means, Spectral Clustering)*

- **Classification**
  - Naive Bayes *(MR only: Logistic Regression, Hidden Markov Models)*

- **Recommenders**
  - Correlated Cross-Occurrence (CCO) - new Multimodal recommender

- **Collaborative filtering**
H2O: Big Data ML

• Open source, in-memory, distributed machine learning and predictive analytics
• H2O’s core code is written in Java.
  – Also supports Python and R
• Data is distributed across the cluster and stored in tabular format
• Distributed Key/Value store for reference data, models and other objects
• ML algorithms are implemented using H2O MapReduce framework and utilize Java Fork-Join framework for multi-threading.
• H2O’s REST API allows access to all the capabilities of H2O from an external program or script via JSON over HTTP
H2O

- HDFS, Spark, S3, Azure Data Lake as **data sources**
- Deploy on bare metal servers or in existing Hadoop and Spark clusters
- Sparkling Water allows to directly integrate H2O with Spark
  1. Transform input data using Spark
  2. Apply H2O to build models
  3. Manipulate results in Spark
- Integratable with Stream processing systems such as Apache Storm
  - Use H2O functionality as Storm Bolts inside Storm topologies

**Introduces AutoML**
- Automatically execute a set of ML algorithms
- Vary the set of their hyperparameters
- Produce a leaderboard of the resulting models
ML Algorithms

**Supervised**
- Cox Proportional Hazards (CoxPH) regression
- Deep Learning (Neural Networks)
- Distributed Random Forest (DRF)
- Generalized Linear Model (GLM) regression
- Gradient Boosting Machine (GBM)
- Naïve Bayes Classifier
- Stacked Ensembles (stacking learners)
- XGBoost (regression and classification)

**Unsupervised**
- Aggregator – record reduction
- Generalized Low Rank Models (GLRM) – Dimensionality reduction
- Isolation Forest – anomaly detection
- K-Means Clustering
- Principal Component Analysis (PCA) – dimensionality reduction
H2O platform

https://www.h2o.ai/products/h2o/#overview
Spark Machine learning library

• A set of scalable machine learning methods implemented in Spark
• Accessible through both RDD and DataFrame interface
  – DataFrame based ML API `spark.ml` is considered primary
  – RDD-based APIs in the `spark.mllib` package is in maintenance mode
Spark ML-Lib functionality

- **Feature manipulation**
  - feature extraction, transformation, selection, dimensionality reduction

- **Machine learning methods**
  - Regression, classification, clustering, etc.

- **Pipelines**

- **Persistence**
  - Models and Pipelines

- **Utilities**
  - Linear algebra, data formats, UDF’s, etc.
  - Statistical Summarizer, Hypothesis testing, Correlation matrices
Feature manipulation

• Feature Extractors
  – TF-IDF, Word2Vec, CountVectorizer, FeatureHasher (hash trick)

• Feature Selectors
  – VectorSlicer, Rformula
  – ChiSqSelector (Pick top features according to a chi-squared test)

• Feature Transformers
  – Tokenizer, n-gram, Normalizer, VectorAssembler

• Locality Sensitive Hashing
  – Dimensionality reduction
  – Use hashing to divide data objects into buckets
  – Making sure similar items „hash“ into the same bucket
# Input data: Each row is a bag of words from a sentence or document.

documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "),),
    ("I wish Java could use case classes".split(" ")),
    ("Logistic regression models are neat".split(" ")),
    ("Logistic regression models are great".split(" ")),],
    ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)

| text                                      | result                                                            |
|------------------------------------------+------------------------------------------------------------------|
| 
|
| [Hi, I, heard, about, Spark]             | [-0.0125114776501822473, -0.041098621767014266, -0.03527139872312546] |
| [I, wish, Java, could, use, case, classes] | [-0.008153195210200335, 0.024885651389403, 0.008441961237362452] |
| [Logistic, regression, models, are, neat] | [0.0387723462561404, -0.020189061760902405, 0.03149386569857598] |
| [Logistic, regression, models, are, great] | [0.02199809551239014, 0.019442501664161685, 0.04460197687149048] |

Source: https://spark.apache.org/docs/latest/ml-features.html#word2vec
### Classification & Regression

#### Classification
- Decision tree classifier
- Random forest classifier
- Linear Support Vector Machine
- Naive Bayes
- Logistic regression
- Binomial logistic regression
- Multinomial logistic regression
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- One-vs-Rest classifier

#### Regression
- Linear regression
- Generalized linear regression
- Available families
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression
Clustering and recommendation

• K-means
• Latent Dirichlet allocation (LDA)
  – Document topic modelling
  – Topics are cluster centers
• Bisecting k-means - Hierarchical clustering
• Gaussian Mixture Model (GMM)
  – Probabilistic: avoid fixing data objects into specific cluster
  – Better when clusters have different „shapes“ and sizes

• Collaborative filtering
  – Recommender systems
  – Predict what users like based on what similar users have liked
K-means example (Iris dataset)

dataset = spark.read.option("inferSchema", True).csv(input_file)
       .toDF("slen", "swidth", "plen", "pwidth", "class")

+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
<table>
<thead>
<tr>
<th>slen</th>
<th>swidth</th>
<th>plen</th>
<th>pwidth</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>Iris-setosa</td>
</tr>
</tbody>
</table>
+-----------------+-----------------+-----------------+-----------------+-----------------+

dataset.select("class").distinct().show()

+---------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
<table>
<thead>
<tr>
<th>class</th>
<th>Iris-virginica</th>
<th>Iris-setosa</th>
<th>Iris-versicolor</th>
</tr>
</thead>
</table>
+---------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
Feature selection using VectorAssembler

assembler = VectorAssembler(
    inputCols=['slen', 'swidth', 'plen', 'pwidth'],
    outputCol='features')

featured = assembler.transform(dataset)
Building and using the K-Means model

```python
kmeans = KMeans().setK(3).setSeed(1)
model = kmeans.fit(featured)
predictions = model.transform(featured)
predictions.show(10, False)
```

<table>
<thead>
<tr>
<th>slen</th>
<th>swidth</th>
<th>plen</th>
<th>pwidth</th>
<th>class</th>
<th>features</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.1, 3.5, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.9, 3.0, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.7, 3.2, 1.3, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.6, 3.1, 1.5, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.0, 3.6, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>Iris-setosa</td>
<td>[5.4, 3.9, 1.7, 0.4]</td>
<td>2</td>
</tr>
<tr>
<td>4.6</td>
<td>3.4</td>
<td>1.4</td>
<td>0.3</td>
<td>Iris-setosa</td>
<td>[4.6, 3.4, 1.4, 0.3]</td>
<td>2</td>
</tr>
<tr>
<td>5.0</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.0, 3.4, 1.5, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.4</td>
<td>2.9</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.4, 2.9, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>Iris-setosa</td>
<td>[4.9, 3.1, 1.5, 0.1]</td>
<td>2</td>
</tr>
</tbody>
</table>
Clustering results

• Lets evaluate how well the classes were distributed among the clusters
• We can apply the Spark DataFrame crosstab operation

predictions.crosstab("class", "prediction").show()

+-------------------+---+---+---+
<table>
<thead>
<tr>
<th>class_prediction</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iris-virginica</td>
<td>14</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>48</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>-------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>
ML Pipelines

- Spark ML operations can be chained into pipelines
- Join VectorAssembler and Kmeans into a single Pipeline

```python
assembler = VectorAssembler(
    inputCols = ['slen', 'swidth', 'plen', 'pwidth'],
    outputCol = 'features')

kmeans = KMeans().setK(3).setSeed(1)

pipeline = Pipeline(stages = [assembler, kmeans])
model = pipeline.fit(dataset)
```
Model persistence

- Models and pipelines can be saved to the filesystem (HDFS) and later loaded from there
- Allows to build models/pipelines ahead of time
- Built models can be migrated to other servers/clusters
- Saving models:

  \texttt{model.save("myKmeansPipeline")}

- Loading saved models:

  \texttt{saved_model = KMeansModel.load("myKmeansPipeline")}
Why use DataFrames for ML?

- Provide a uniform API across multiple languages
  - Java, Scala, Python,
  - R not so much
- More user-friendly than RDDs.
  - Convenient for users who have used dataframes in other languages
- Tabular format is natural for feature selection and manipulation
- Spark Datasources streamline data importing (CSV, JSON, Kafka streams)
- Tungsten and Catalyst optimizers can take advantage of stricter tabular data schemas
  - **Catalyst**: SQL query-to-code optimizer (Logical and Physical plans)
  - **Tungsten**: Off-Heap Memory Management, cache-aware computations, whole-stage code generation/merging
Distributed R with Spark
SparkR

- Spark interface for scaling R computations
- Based on DataFrame API
  - import SparkR API
  - Convert DataFrames between R and Spark
  - Apply Spark DataFrame operations and R UDF’s in parallel
- Spark R DataFrame implements
  - R dataframe operations
  - Spark DataFrame operations (including also SQL and MLlib capabilities)
- Usable directly from R Studio and Shiny
  - Can define the location of the remote cluster
- Rather different API from Spark Java, Scala and Python
Typical use cases

• Managing large amount of data
  – Load in a lot of data with Spark DataFrame API
  – Aggregate, manipulate data with Spark DataFrame and SparkR operations
  – Convert intermediate result into R DataFrame
  – Apply R for statistics and generating graphs

• Speeding up computationally heavy operations
  – Load in and manipulate data with R
  – Convert R DataFrame into Spark
  – Use Spark to accelerate more computationally heavy operations
    • Apply UDF on each row of data
    • Apply Distributed ML, compute correlation matrices, etc.
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))

sparkR.session(master = "local[*]", sparkConfig = list(spark.driver.memory = "2g"))

# Create a SparkDataFrame based using the faithful dataset from R.
df <- as.DataFrame(faithful)

# We use the 'n' operator to count the number of times each waiting time appears
head(summarize(groupBy(df, df$waiting), count = n(df$waiting)))

<table>
<thead>
<tr>
<th>waiting</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>69</td>
</tr>
</tbody>
</table>

# Sort the output from the aggregation to get the most common waiting times
waiting_counts <- summarize(groupBy(df, df$waiting), count = n(df$waiting))

head(arrange(waiting_counts, desc(waiting_counts$count)))

<table>
<thead>
<tr>
<th>waiting</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>83</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
</tr>
</tbody>
</table>
SparkR ML example

```
sparkR.session(appName = "SparkR-ML-kmeans-example")

#Load in data as an R DataFrame
r <- as.data.frame(Titanic)

training <- createDataFrame(r)
df_list <- randomSplit(training, c(7,3), 2)
kmeansDF <- df_list[[1]]
kmeansTestDF <- df_list[[2]]

# Fit a k-means model with spark.kmeans
kmeansModel <- spark.kmeans(kmeansDF, ~ Class + Sex + Age + Freq, k = 3)
# Model summary
summary(kmeansModel)

# Get fitted result from the k-means model
head(fitted(kmeansModel))

# Prediction
kmeansPredictions <- predict(kmeansModel, kmeansTestDF)
```

Summary of the course:

The Big Picture
Has Big Data Ecosystem become a too complex landscape of tools?
The 2018 Big Data & AI Landscape

http://mattturck.com/bigdata2018/
Big Data ecosystem

Has the Big Data Ecosystem become a too complex landscape of tools?

• **Yes?**
  • Too many options!
  • Hard to choose the optimal solution for a specific use case

• **No?**
  • Core is still relatively small
  • Interoperability is reasonable
  • Java and SQL as the main languages
Core of Big Data ecosystem

• Main distributed File systems:
  – HDFS
  – Alternatives: S3 and NFS

• Main distributed computing engines:
  – **Batch**: MapReduce, Spark, Flink
  – **Stream**: Kafka, Storm, Spark, NiFi

• Main bundle distributions
  – Cloudera, Hortonworks and MapR
Distributed computing engines

• **Hadoop MapReduce**
  – *Big Data* expert
  – Everything on HDD

• **Apache Spark**
  – *Not so Big Data*
  – In-Memory processing
  – Jack of ALL trades

• **Kafka**
  – Data stream management
  – Data integration

• **Apache Flink**
  – High performance stream applications

• **Apache NiFi**
  – Data pipelines
  – Data stream integration between systems

• **Apache Storm**
  – Real time event-based stream processing
Don't dive too deep

• **Don't lock** the choice of tools and technologies
  – Start from the actual data and use cases
  – Choose technologies that support the actual need

• Consider the **future requirements**
  – Will data fit into the memory after: launch, X years, expanding to other regions
  – Are the selected tools developed actively?

• Avoid vendor and technology **lock-in**
  – Avoid technologies that lock you into specific tools and programming languages

• Prefer technologies that **integrate well** with other solutions and core technologies
Why we focused on Spark?

• Is being **Jack of all trades** a **Positive** or a **Negative** trait?
• Many Hadoop ecosystem tools have started using Spark as the main alternative to MapReduce engine
  – Apache Pig, Apache Mahout, ...
• Most Spark extensions are built-into Spark core
• Still somewhat confusing when to use RDD and when to use DataFrame
  – DataFrame API seems to have won the hearts of the Spark community
Why we focused on Spark?

• When data does not fit into memory?
  – Use Spark streaming to crunch incoming Big Data smaller

• Need more efficient real-time Stream processing
  – Apache Storm
  – Apache Flink
  – Apache Kafka ( + Spark)

• Need additional ML methods
  – H20 can be directly integrated with Spark

• Spark community is very active
Apache Spark activity

• Last week, from 6 til 13 December:
  – 28 authors have pushed 47 commits
  – 193 files have changed
  – 3,505 additions and 1,818 deletions.

• Weekly contributions over time:
Conclusions

• Use MapReduce, Pig or Hive when data does not fit into the memory of the cluster
• Otherwise Spark may be the most optimal solution
  – It is much faster in general
  – Prototyping is convenient
  – Many built-in and external extensions
  – Many UDF's which can be included on the fly
  – Spark is constantly evolving
• Except for 24/7 streaming applications
  – Use Storm, Flink, or Kafka (+ Spark)
• Many domain specific solutions support or integrate with both MapReduce and Spark

Whether you choose MapReduce, Spark, Pig Latin, or SQL –
Everything is automatically parallelized in the background
and can be executed across clusters of computers
Examination
Course grade formula

• Final grade consists of three components:
  1. Written exam – 50%
  2. Labs – 45%
  3. Participation in the lectures - 5%

• NB! You need to collect at least 50% in each grade component to pass the course!
Examination

• Examination times
  – Friday 04. January 12:00 - 15:00 (Ü17 – 219, 220)
  – Monday 07. January 10:00 - 13:00 (Ü17 – 219)

• Location is Ülikooli 17, rooms 219 or 220

• Exam lasts 3 hours!

• Most students currently registered to the exam on 4th
Preparation for Examination

• Video and slides will be online for all lectures
• Previous course in the series
  – Large-scale Data Processing on the Cloud
  – One of the earlier exam papers is kept online
    • But covers only 50% of topics

• References will be provided at the end of the slides for supportive reading
Thats All

• Next weeks practice session is
  – Machine Learning in Apache Spark (Python)

• Exams:
  – Friday 04. January 12:00 - 15:00
  – Monday 07. January 10:00 - 13:00
References

• Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. SIGMOD 2010.
• Rick Cattell, Scalable SQL and NoSQL Data Stores. SIGMOD Rec, ACM, December 2010.