Big Data Technologies

Agenda

- Spark ML concepts
- Spark Core (Low Level APIs)

Advanced Analytics

- Analysis vs Analytics
  - Analysis --> Past data -- Understading data, Summarizing data, Visualizing data.
    - Python: Pandas, Numpy, Matplotlib, etc.
    - R: Frames, Stats, Charts, etc.
    - Excel: Ordering, Filtering, Pivot table/charts, etc.
    - PowerBI: Charts, Reports, etc.
  - Analytics --> Future/Predictions
    - Machine Learning --> Single system --> Languages: Python, R, C++, etc
    - Machine Learning --> Distributed systems --> Mahaout on Hadoop (outdated), Spark ML, etc.

Analytics

- Analytics refer to various techniques to solve core problem of deriving insights and making predictions/recommendations.
- Most common tasks are:
  - Supervised Learning
    - most common
    - using historical data train model
      - data have label (dependent variable)
      - data have features
    - Includes
      - classification: predict disease, predict purchase or not, classify images, ...
      - regression: predict sales, predict viewer count, ...

- Recommendation Engines
  - movie recommendation, product recommendation, ...
- Unsupervised Learning
  - find pattern or discover underlying struct in data
  - clustering, anomaly detection, topic modeling, ...
- Graph Analytics
  - based on graph data struct algos
  - fraud detection, classficiation, recommendation, find patterns in social network, ...

Analytics Process

1. Collect relevant data.
2. Clean & inspect the data -- EDA.
3. Feature Engg -- Extract features into numerical vectors
4. Build ML model using portion of data (training set).
5. Evaluate model using portion of data (test set).
6. Serve ML model to predict, recommend, ...

Apache Spark - Spark ML

1. Collect relevant data.
   - Spark can get data from any source -- HDFS, S3, RDBMS, NoSQL, ...
   - Spark can get live data (streaming) -- Kafka, Flume, Kinesis, ...
2. Clean & inspect the data (EDA).
   - Spark can do Regex, Filtering, Corrections, Enriching, etc.
   - Spark can do Batch processing and/or Streaming processing
3. Feature Engg -- Extract features into numerical vectors
   - Spark has Transformers and Estimators.
4. Build ML model using portion of data (training set).
   - Spark ML supports Supervised ML, Unsupervised ML, Recommendations, NLP, etc.
5. Evaluate model using portion of data (test set).
   - Spark does accuracy check using Evaluators.

6. Serve ML model to predict, recommend, ...
    - Allows to export model in various formats like PMML, Pickle, etc.

Spark ML components

- Includes data collection, cleaning, feature engg, training, evaluating large scale supervised & unsupervised models.
- Advantages/Applications
    - Preprocessing & feature engg.
    - Building models for huge training data.
- (High Level) Components
    - Transformers -- transform()

```
# example
vectAssembler = VectorAssembler()\
    .setInputCols(['Age', 'Salary', 'GenderIndexed'])\
    .setOutputCol('features')
df3 = vectAssembler.transform(df2)
```

    - Estimators -- fit() + transform()

```
# example
genderIndexer = StringIndexer()\
    .setInputCol('Gender')\
    .setOutputCol('GenderIndexed')
df2 = genderIndexer.fit(df1)\
    .transform(df1)
```

    - Models -- fit(training) + transform(test)

```
model = LogisticRegression()\
            .setFeaturesCol('features')\
            .setLabelCol('label')\
            .fit(train_df)
predictions_df = model.transform(test_df)
```

- Evaluators -- checking accuracy of model on test data

```
# example
model = LogisticRegression().fit(train)
predictions = model.transform(test)

accuracy = BinaryClassificationEvaluator()\
    .evaluate(predictions)
print('Model Accuracy: ', accuracy)
```

- Pipelines -- stages to build model (includes Transformers & Estimators).

```
# example
mlPipeline = Pipeline()\
    .setStages([genderIndexer, vectAssembler, mlModel])
```

- (Low Level) Components
  - Vectors -- Sparse or Dense vectors.

```
from pyspark.ml.linalg import Vectors
denseVec = Vectors.dense(1.0, 2.0, 3.0)
# denseVec -- (1.0, 2.0, 3.0)
```

```
from pyspark.ml.linalg import Vectors
size = 3            # vector size
idx = [1, 2]        # locations of non-zero elements in vector
values = [2.0, 3.0] # non-zero elements
sparseVec = Vectors.sparse(size, idx, values)
# sparseVec -- (0.0, 2.0, 3.0)
```

```
from pyspark.ml.linalg import Vectors
idxIn = spark.createDataFrame([
    (Vectors.dense(1, 2, 3), 1),
    (Vectors.dense(2, 5, 6), 2),
    (Vectors.dense(1, 8, 9), 3)
]).toDF("features", "label")
```

Apache Spark - Spark Core

Spark Installation -- Local Mode

- Download "spark-x.y.z-bin-hadoop3.tgz" and extract it.
- In ~/.bashrc

```
export SPARK_HOME=/path/of/spark-x.y.z-bin-hadoop3
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
```

- terminal> which spark-shell
- terminal> spark-shell --master local

```

```

```
val file=sc.textFile("/home/nilesh/setup/spark-3.5.1-bin-hadoop3/LICENSE")

val lines=file.map(line=>line.toLowerCase())

val words=lines.flatMap(line=>line.split("[^a-z]"))

val word1s=words.map(word=>(word,1))

val wordcounts=word1s.reduceByKey((a,x)=>a+x)

val capwordcounts=wordcounts.map(wc=>(wc._1.toUpperCase(), wc._2))

val result=capwordcounts.collect()

capwordcounts.toDebugString

:quit
```

Word Count Program -- Explanation

- Input -- rdd1 (file)

```
Red green blue
Red red green
green Blue black
green Green blue
```

- Lower Case -- rdd2 (lines)

```
red green blue
red red green
```

```
green blue black
green green blue
```

- Split -- flatMap() -- rdd3 (words)

```
red
green
blue
red
red
green
green
blue
black
green
green
blue
```

- Add one -- rdd5 (word1s) -- Key-Value pair RDD (RDD of Tuple2)

```
(red,1)
(green,1)
(blue,1)
(red,1)
(red,1)
(green,1)
(green,1)
(blue,1)
(black,1)
(green,1)
(green,1)
(blue,1)
```

- Group by word and Sum ones -- reduceByKey -- rdd6 (wordcounts)
  - Grouping by Key

```
(black, [1])
(blue, [1,1,1])
(green, [1,1,1,1,1])
(red, [1,1,1])
```

  - Reduce on each group --> (a,b) => a + b
    - "1, 1," 1, 1, 1
      - 1 + 1 = 2
    - 1, 1, "1," 1, 1
      - 2 + 1 = 3
    - 1, 1, 1, "1," 1
      - 3 + 1 = 4
    - 1, 1, 1, 1, "1"
      - 4 + 1 = 5
  - Result of reduce

```
(black, 1)
(blue, 3)
(green, 5)
(red, 3)
```

- Capitalize the word --> rdd7 (capwordcounts)

```
(BLACK, 1)
(BLUE, 3)
```

```
(GREEN, 5)
(RED, 3)
```

RDD From List

```
```scala
# list --> parallelize() --> rdd
val rdd1 = sc.parallelize(Array(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16), 4)
rdd1.getNumPartitions

val rdd2 = rdd1.map(num => num * num)
rdd2.getNumPartitions

rdd2.saveAsTextFile("/tmp/result")
# terminal> ls /tmp/result
```
```

Local Mode (PySpark Installation)

- pyspark is a python package designed to develop & test pyspark applications (in local mode).
- terminal> python3 -m pip install pyspark
- In ~/.profile

```
export PYSPARK_PYTHON=python3
export PYSPARK_DRIVER_PYTHON=python3
export SPARK_HOME=$HOME/.local/lib/python3.8/site-packages/pyspark
export PATH=$HOME/.local/bin:$PATH
```

- terminal> pyspark --master local
- OR

- terminal> pyspark

```
file = sc.textFile("/home/nilesh/setup/bigdata/spark-3.3.1-bin-hadoop3/LICENSE")

lines = file.map(lambda line: line.lower())

words = lines.flatMap(lambda line: line.split())

word1s = words.map(lambda word: (word,1))

wordcounts = word1s.reduceByKey(lambda acc,cnt: acc + cnt)

result = wordcounts.collect()

print(result)
```

PySpark Development (PyCharm/VSCode)

- PyCharm -> New Project
- Select project location
- Existing interpreter -> Python3.x
- Create Python file (hello.py)

```
from pyspark import SparkConf
from pyspark import SparkContext
conf = SparkConf().setAppName("Demo01").setMaster("local")
sc = SparkContext(conf=conf)
file = sc.textFile("/home/nilesh/setup/bigdata/spark-3.3.1-bin-hadoop3/LICENSE")
lines = file.map(lambda line: line.lower())
words = lines.flatMap(lambda line: line.split())
word1s = words.map(lambda word: (word,1))
wordcounts = word1s.reduceByKey(lambda acc,cnt: acc + cnt)
```

```
result = wordcounts.collect()
print(result)
```

Single Node Cluster (Spark Standalone)

- Download & extract spark-x.y.z-bin-hadoop3.tgz.
- In ~/.bashrc

```
export SPARK_HOME=$HOME/spark-x.y.z-bin-hadoop3
export PATH=$SPARK_HOME/bin:$SPARK_HOME/sbin:$PATH
```

- In $SPARK_HOME/conf/spark-env.sh

```
export SPARK_MASTER_HOST=localhost
export SPARK_LOCAL_IP=localhost
```

- In conf/workers.

```
localhost
```

- In $SPARK_HOME/conf/spark-defaults.conf

```
spark.master            spark://localhost:7077
```

- Start master & slaves.
  - terminal> start-master.sh

    - terminal> start-workers.sh
    - terminal> jps
- Using cluster:
    - terminal> pyspark --master spark://localhost:7077
    - terminal> spark-submit

Multi-node cluster (Spark Standalone)

- step 1: Follow these steps on all machines.
    - Create a new user "hduser".
    - Download "spark-x.y.z-bin-hadoop3.tgz" and extract it in $HOME.
- step 2. Change hostname of all machines (as appropriate). In Ubuntu this can be done using hostnamectl.
    - `sudo hostnamectl set-hostname master`
- step 3. In /etc/hosts make entry of master and workers/slaves on all machines.

`sudo vim /etc/hosts`

```
192.168.56.10 master
192.168.56.11 worker1
192.168.56.12 worker2
192.168.56.13 worker3
```

- step 4. Ensure that all machines are running and connect to each other using "ping". Try commands from all machines.

```
ping master
ping worker1
ping worker2
ping worker3
```

- step 5. Enable password-less login of master on all slaves. Follow these steps on master.

```
ssh-keygen -t rsa -P ""

ssh-copy-id $USER@master

ssh-copy-id $USER@worker1

ssh-copy-id $USER@worker2

ssh-copy-id $USER@worker3
```

- step 6. On Master machine, conf/workers make entries of all workers.

```
worker1
worker2
worker3
```

- step 7. In all machines, conf/spark-defaults.conf

```
spark.master spark://master:7077
```

- step 8. On master machine, set SPARK_LOCAL_IP & SPARK_MASTER_HOST to be set (in conf/spark-env.sh) to the IP address of network.

```
export SPARK_LOCAL_IP=master
export SPARK_MASTER_HOST=master
```

- step 9. On each slave machine, set SPARK_LOCAL_IP (in conf/spark-env.sh) to the IP address of network.

```
export SPARK_LOCAL_IP=workerX
```

- step 10. From master machine.

```
terminal> start-master.sh

terminal> start-workers.sh
```

  - Check browser UI - http://master:8080/

- step 11. Use spark cluster

```
terminal>pyspark --master spark://master:7077
# OR
terminal> spark-submit --master spark://master:7077 --deploy-mode client app.py
```

Spark RDD

- Resilient Distributed Dataset
  - Resilient -- Can be recovered (recomputed - from RDD Lineage) in case of failures
  - Distributed -- Divided into multiple Partitions in RAM of multiple nodes in Spark cluster
  - Dataset -- Set/Collection of Data
- RDD characteristics
  - Immutable -- Each operation will create new RDD. Existing RDD is not modified.
  - Lazily evaluated -- RDD execution will begin only if action operation is specified.
  - Resilient -- Can be recomputed.

Spark RDD – creation

- sc.parallelize(collection, partitions)
  - convert collection into rdd with given partitions
- sc.textFile(path)
  - hdfs or local or s3 file or directory
- sc.wholeTextFiles(path)
  - hdfs or local or s3 directory, one file = one record
- sc.binaryRecords(path, recLen)
  - hdfs or local or s3 file or directory, recLen bytes = one record
- sc.wholeBinaryFiles(path)
  - hdfs or local or s3 directory, one file = one record
- sc.hadoopFile(path, inputFormat, …)
  - hdfs file or directory, number of partitions = number of input splits

RDD Operations

- Transaformations
  - RDD --> Transaformation --> New RDD
  - Narrow transformations
    - A partition of new RDD is computed from single partition of source RDD
  - Wide/Shuffle transformations
    - Partition of new RDD is computed from multiple partitions of source RDD
    - These transformations cause shuffling data across partitions.
- Actions
  - RDD --> Action --> Result (Non-RDD)
- Transformations: Returns RDD
  - distinct()
  - filter()
  - map()
  - flatMap()
  - sortBy(keyValue, ascending, numOfPartitions)

    - sortByKey()
    - pipe()
    - keyBy()
    - countByKey()
    - mapValues()
    - groupByKey()
    - aggregateByKey()
    - cogroup()
    - zip()
    - join() – joins two RDDs by key.
    - reduceByKey()
- Actions: Returns non-RDD
    - count()
    - countApprox()
    - reduce()
    - countByValue()
    - first()
    - max()
    - min()
    - collect()
    - take() – collect n elements
    - takeOrdered()
    - top()
    - saveAsTextFile()
    - saveAsObjectFile()
    - lookup() – lookup by key