Dec-2021

ANALYSIS OF WEATHER USING SPARK

A PROJECT REPORT

Submitted by

BL.EN.U4AIE19018 G.Y.SAI VARUN

BL.EN.U4AIE19021 G. SAI SURYA TEJA

BL.EN.U4AIE19022 G.B.R.RAVI TEJA

19AIE304 BIG DATA AND DATABASE MANAGEMENT

AMRITA SCHOOL OF ENGINEERING, BANGALORE

AMRITA VISHWA VIDYAPEETHAM

BANGALORE 560 035

ABSTRACT

Changes in humidity have substantial knock-on impacts at both the global and local level, because of its impact on increase of greenhouse gases. Air Quality is a major problem these days across the globe, in this project an analysis is done on the weather using air quality data. “Sofia air quality dataset" is being used in this project. Spark Streaming and DataFrames are used to stream the data, analyze it and spark’s ML library is used to Build and Train the Machine Learning Model Which Predicts the Humidity.

TABLE OF CONTENTS

INTRODUCTION

Air pollution is increasing every year which is growing rapidly, which is decreasing the quality of the air. As it is getting worse, citizens are mainly concerned about the health. So, for focusing on the decreasing the pollution, it is necessary to make an analysis of the parameters like temperature, humidity, pressure etc. By using the Sofia Air Quality Data Set, A Spark analysis and data prediction system is being designed where this model can-do real-time prediction using ML Algorithms like Linear Regression and Decision Tree Algorithm, by predicting the humidity, which gives a direct inference about greenhouse gases and global warming. Since, Air Quality Data is very large implementing this using Spark will be more efficient in terms of speed, since it follows parallelism and live weather data will also be coming, which Spark Streaming Would take care off.

Apache spark:

- Apache Spark is a cluster computing platform designed to be fast and general purpose.
- On the speed side, Spark extends the popular MapReduce model to efficiently sup-port more types of computations, including interactive queries and stream processing.
- Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours.
- One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also more efficient than MapReduce for complex applications running on disk.

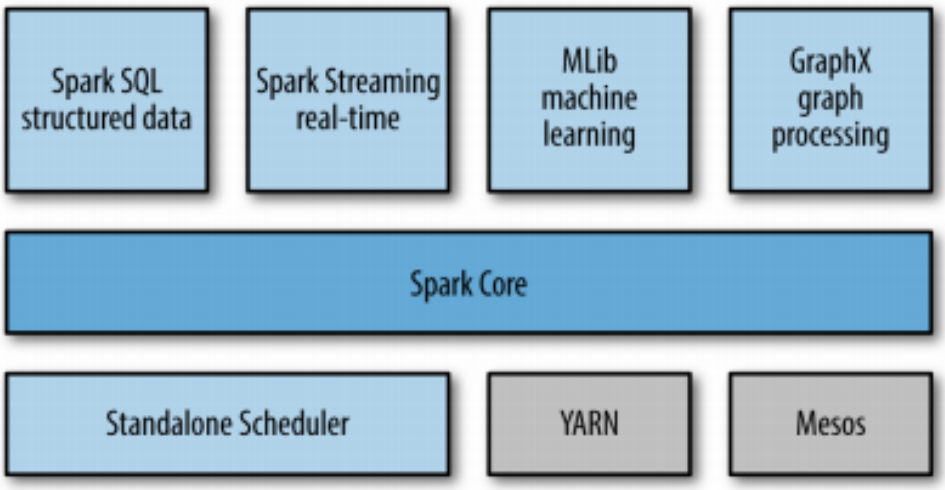


Fig-1: Spark Components

Spark Core:

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed data- sets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL:

Spark SQL is Spark's package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open-source data warehouse tool. Spark SQL was added to Spark in version 1.0.

Shark was an older SQL-on-Spark project out of the University of California, Berkeley, that modified Apache Hive to run on Spark. It has now been replaced by Spark SQL to provide better integration with the Spark engine and language APIs.

Spark Streaming:

Spark Streaming is a Spark component that enables processing of live streams of data. Examples of data streams include logfiles generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real time. Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability as Spark Core.

MLlib:

Spark comes with a library containing common machine learning (ML) functionality, called MLlib. MLlib provides multiple types of ML algorithms, including classification, regression, clustering, and collaborative filtering, as well as sup- porting functionality such as model evaluation and data import. It also provides some lower-level ML primitives, including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.

Pyspark:

Apache Spark is written in Scala programming language. PySpark has been released in order to support the collaboration of Apache Spark and Python, it actually is a Python API for Spark.

In addition, PySpark, helps you interface with Resilient Distributed Datasets (RDDs) in Apache Spark and Python programming language. This has been achieved by taking advantage of the Py4j library

Fig-2: Formation of PySpark

Advantages of using PySpark:

- Python is very easy to learn and implement.
- It provides simple and comprehensive API.
- With Python, the readability of code, maintenance, and familiarity is far better.
- It features various options for data visualization, which is difficult using Scala or Java.

Linear Regression:

The most fundamental and widely used type of predictive analysis is linear regression. The goal of regression is to look at two things: Is it possible to forecast an outcome (dependent) variable using a set of predictor variables? Which variables in particular are significant predictors of the outcome variable, and how do they influence the outcome variable (as indicated by the size and sign of the beta estimates)? These regression estimations are used to illustrate how one dependent variable interacts with one or more independent variables. The simplest version of the regression equation with one dependent and one independent variable is y = c + b*x, where y represents the estimated dependent variable score, c represents the constant, b represents the regression coefficient, and x represents the independent variable score.

The Variables' Names The dependent variable in a regression has several names. It may be called an outcome variable, criterion variable, endogenous variable, or regress and. The independent variables can be called exogenous variables, predictor variables, or regressors. Three major uses for regression analysis are determining the strength of predictors, forecasting an effect, and trend forecasting.

Decision tree:

Decision tree is the most powerful and popular tool for classification and prediction. It is a like a flowchart like tree structure in which each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node holds a class label. Decision trees are able to generate understandable rules. They can perform classification without requiring computation. They are able to handle both continuous and categorical variables. They provide clear indication of which fields are most important for prediction or classification. Decision trees are part of the foundation for machine learning. Although they are quite simple, they are very flexible and popup in a very wide variety of situations.

Construction:

A tree can be learned by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node all has the same value of the target variable. It also be completed when splitting no longer adds value to the predictions. This construction does not require any domain knowledge or parameter setting and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. It has a good accuracy. It also has a typical inductive approach to learn knowledge on classification.

Representation:

Decision trees classify the instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node then moving down the tree branch corresponding to the value of the attribute. This process is then repeated for the subtree rooted at the new node.

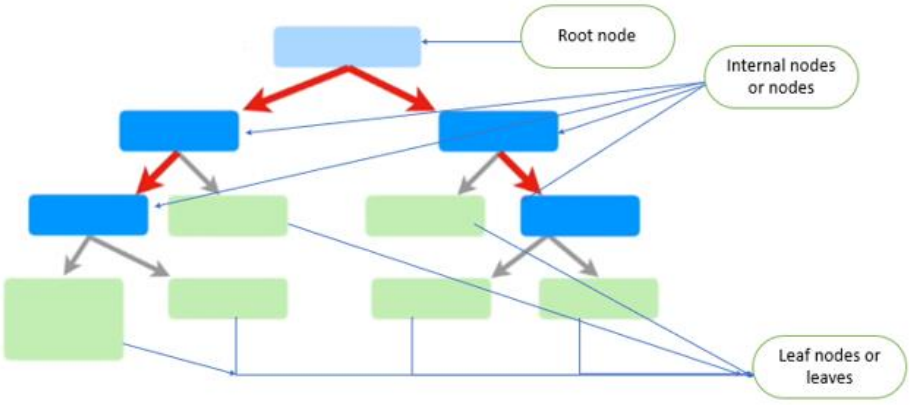


Fig-3:Representation of Decision Trees

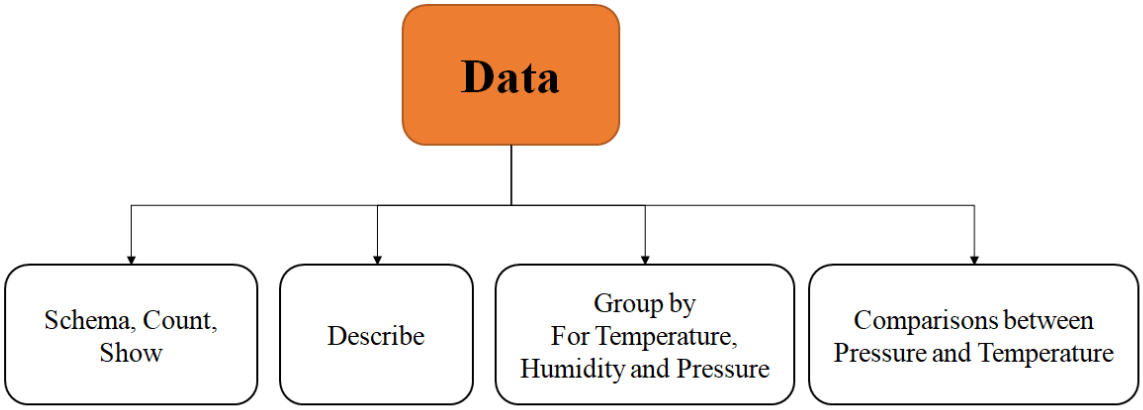
The very top of the tree is called root node. The dark-coloured nodes are called as just nodes or the internal nodes. Internal nodes are having arrows pointing to them (red arrows) and arrows pointing away from them(grey). The nodes which are in green are denoted as leaf nodes and they have just arrows pointing to them and they don't have arrows pointing away from them.

Explanation:

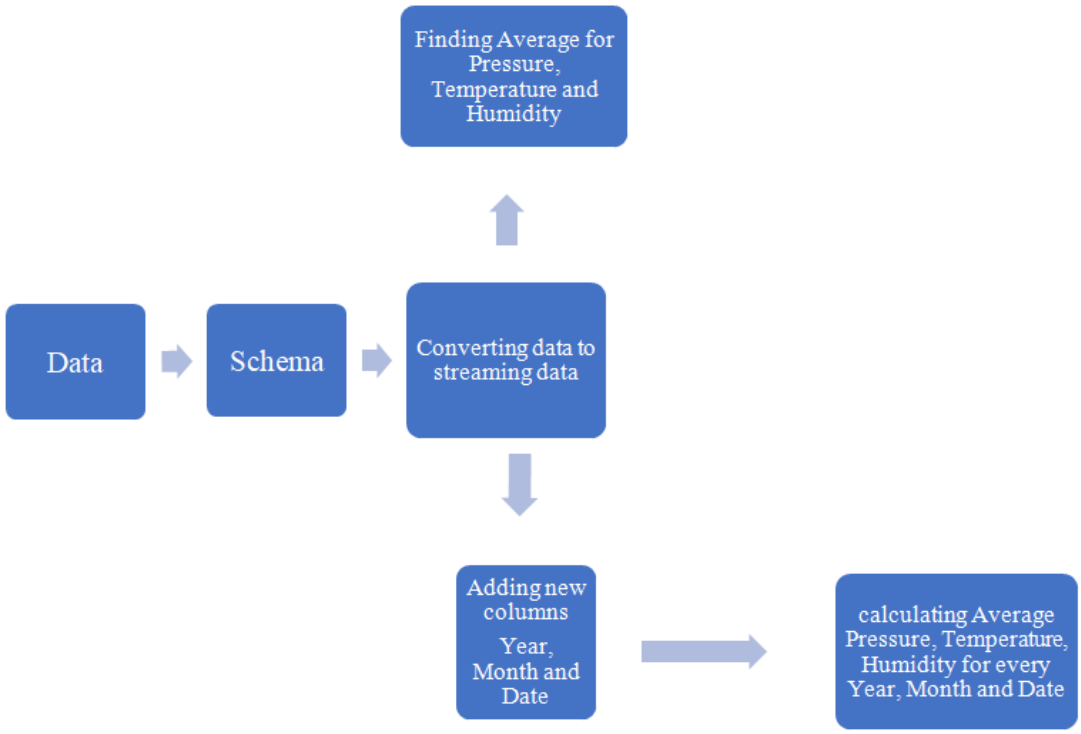
In this explanation, we will look into how we go from a raw table of data to a decision tree by creating a tree that uses chest pain, good blood circulation and blocked artery status to predict whether or not a patient has heart disease. Let's consider the data set. We will be using this data seta and construct the decision tree.

PROPOSED SYSTEM ARCHITECHURE

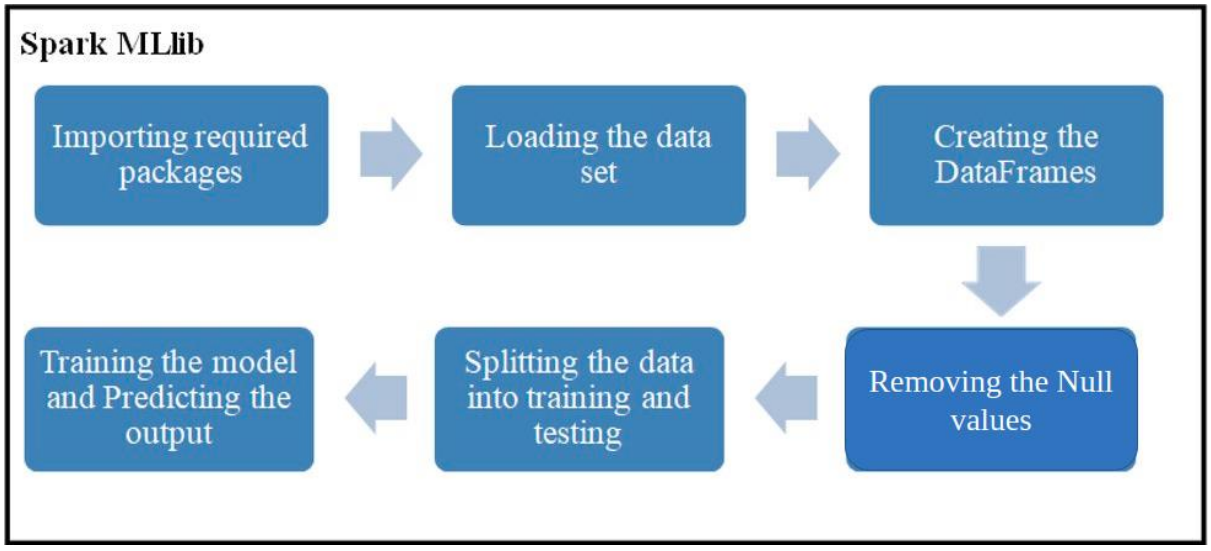
Spark DataFrames:



Structure Streaming:



Machine Learning:



Packages used in our project:

Spark SQL:

Spark's interface for working with structured and semi structured data. Structured data is any data that has a schema—that is, a known set of fields for each record. When you have this type of data, Spark SQL makes it both easier and more efficient to load and query. Spark SQL provides three main capabilities:

- It can load data from a variety of structured sources (e.g., JSON, Hive, and Parquet).
- It lets you query the data using SQL, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC), such as business intelligence tools like Tableau.
- When used within a Spark program, Spark SQL provides rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL, and more. Many jobs are easier to write using this combination.

Spark SQL brings native support for SQL to Spark and streamlines the process of querying data stored both in RDDs (Spark's distributed datasets) and in external sources. Spark SQL also includes a cost-based optimizer, columnar storage, and code generation to make queries fast. At the same time, it scales to thousands of nodes and multi-hour queries using the Spark engine, which provides full mid-query fault tolerance, without having to worry about using a different engine for historical data.

To implement these capabilities, Spark SQL provides a special type of RDD called SchemaRDD. A SchemaRDD is an RDD of Row objects, each representing a record.

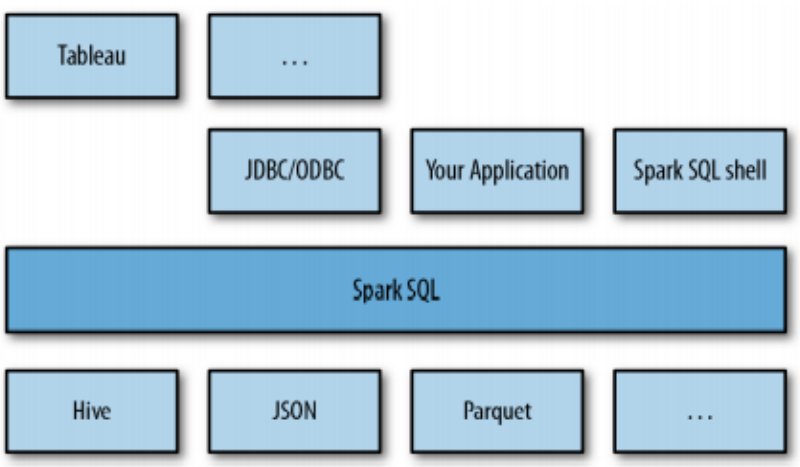


Fig-4: usage of sparkSQL

MLlib:

MLlib's design and philosophy are simple: it lets you invoke various algorithms on distributed datasets, representing all data as RDDs. MLlib introduces a few data types (e.g., labeled points and vectors), but at the end of the day, it is simply a set of functions to call on RDDs.

For example, to use MLlib for a text classification task (e.g., identifying spammy emails), you might do the following:

- Start with an RDD of strings representing your messages.
- Run one of MLlib's feature extraction algorithms to convert text into numerical features (suitable for learning algorithms); this will give back an RDD of vectors.
- Call a classification algorithm (e.g., logistic regression) on the RDD of vectors; this will give back a model object that can be used to classify new points.
- Evaluate the model on a test dataset using one of MLlib's evaluation functions.

Vector Assembler:

Vector Assembler is a transformer that combines a given list of columns into a single vector column. It is useful for combining raw features and features generated by different feature transformers into a single feature vector, in order to train ML models like logistic regression and decision trees.

Vector Assembler accepts the following input column types: all numeric types, Boolean type, and vector type. In each row, the values of the input columns will be concatenated into a vector in the specified order.

Pipeline:

ML Pipelines provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines.

Pipeline components:

Transformers:

A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a method transform(), which converts one DataFrame into another, generally by appending one or more columns.

For example:

A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.

A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.

Estimators:

An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data. Technically, an Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer. For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a Logistic Regression Model, which is a Model and hence a Transformer.

A Pipeline is specified as a sequence of stages, and each stage is either a Transformer or an Estimator. These stages are run in order, and the input DataFrame is transformed as it passes through each stage. For Transformer stages, the transform() method is called on the DataFrame.

For Estimator stages, the fit() method is called to produce a Transformer (which becomes part of the PipelineModel, or fitted Pipeline), and that Transformer's transform() method is called on the DataFrame. We illustrate this for the simple text document workflow. The figure below is for the training time usage of a pipeline.

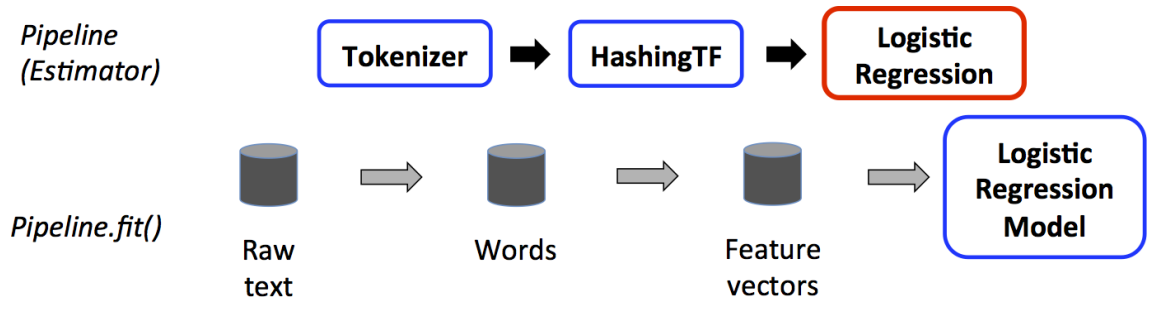


Fig-5: Training time usage of a Pipeline

Above, the top row represents a Pipeline with three stages. The first two (Tokenizer and HashingTF) are Transformers (blue), and the third (LogisticRegression) is an Estimator (red).The bottom row represents data flowing through the pipeline, where cylinders indicate

DataFrames. The Pipeline. fit() method is called on the original DataFrame, which has raw text documents and labels. The Tokenizer. transform() method splits the raw text documents into words, adding a new column with words to the DataFrame. The HashingTF. transform() method converts the words column into feature vectors, adding a new column with those vectors to the DataFrame. Now, since LogisticRegression is an Estimator, the Pipeline first calls Logistic Regression. fit() to produce a Logistic Regression Model. If the Pipeline had more Estimators, it would call the Logistic Regression Model's transform() method on the DataFrame before passing the DataFrame to the next stage.

A Pipeline is an Estimator. Thus, after a Pipeline's fit() method runs, it produces a PipelineModel, which is a Transformer. This PipelineModel is used at *test time;* the figure below illustrates this usage.

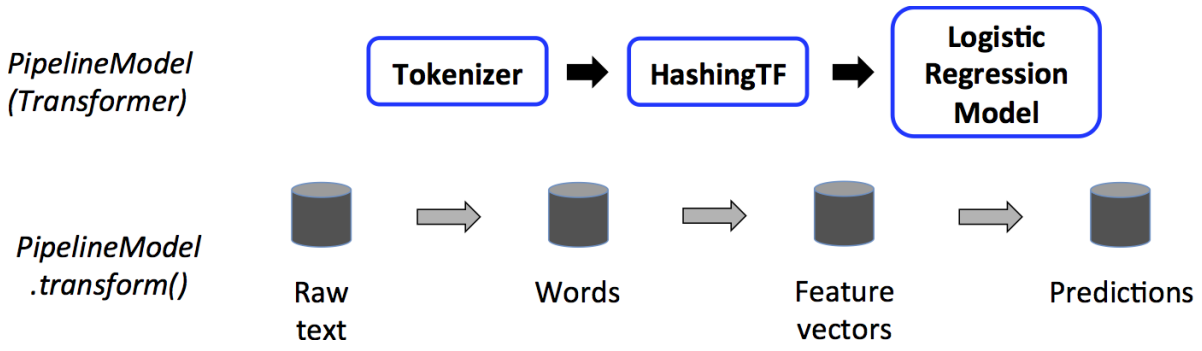


Fig-5:illustrates this usage

In the figure above, the PipelineModel has the same number of stages as the original Pipeline, but all Estimators in the original Pipeline have become Transformers. When the PipelineModels transform() method is called on a test dataset, the data are passed through the fitted pipeline in order. Each stage's transform() method updates the dataset and passes it to the next stage.

Pipelines and PipelineModels help to ensure that training and test data go through identical feature processing steps.

RESULTS

Spark DataFrames:

```
scala> val df = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/Sofia Air quality/*")
df: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> df.columns
res0: Array[String] = Array(_c0, sensor_id, location, lat, lon, timestamp, pressure, temperature, humidity)

scala> df.count
res1: Long = 97288452

scala> df.printSchema()
root
 |-- _c0: integer (nullable = true)
 |-- sensor_id: integer (nullable = true)
 |-- location: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- lon: double (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- pressure: double (nullable = true)
 |-- temperature: double (nullable = true)
 |-- humidity: double (nullable = true)
```

```
scala> df.groupBy("sensor_id").count().show()
+---------+------+
|sensor_id| count|
+---------+------+
|     5300|381427|
|     6466|359282|
|     6393|365571|
|    10462|289243|
|    12626|139362|
|    10798|195936|
|    17837| 75240|
|    20624| 97686|
|     5907|354623|
|     6934| 58959|
|    10840|276968|
|    19067|106969|
|     4477|394828|
|     4799|270126|
|     9721|278561|
|     6678|319907|
|     9867|229453|
|     8085|272731|
|    10022| 81354|
|    11468|221245|
+---------+------+
only showing top 20 rows
```

```
scala> df.groupBy("location").count().show()
+--------+------+
|location| count|
+--------+------+
|    1025|419594|
|    2580|232963|
|   12006| 70804|
|    1139|381552|
|    3488|339798|
|    7281|198803|
|    3098|284536|
|    1884| 74900|
|    4684|110261|
|   11630| 80249|
|     879|227591|
|    2249|177805|
|    3377|342827|
|    3834|125802|
|    2247|356269|
|    2914|282274|
|    7109|149656|
|    3242|330228|
|    5140| 26996|
|    6378|139362|
+--------+------+
only showing top 20 rows
```

```
scala> df.groupBy("lon").count().show()
+------------------+-------+
|               lon|  count|
+------------------+-------+
|            23.336|1100277|
|       23.36456867|  45093|
|23.326999999999998| 387501|
|23.305999999999997| 505021|
|             23.35| 749998|
|            23.259| 412060|
|23.249000000000002| 470520|
|            23.325| 242393|
|23.243000000000002| 185508|
|            23.246| 323705|
|            23.465| 231929|
|23.284000000000002|1826012|
|           23.2932|  48849|
|            23.476| 276427|
|23.340999999999998| 884559|
|            23.344|1137525|
|23.316999999999997| 539247|
|            23.266| 403815|
|            23.256|1214356|
|            23.255| 579161|
+------------------+-------+
only showing top 20 rows
```

```
scala> df.groupBy("lat").count().show()
+------------------+-------+
|               lat|  count|
+------------------+-------+
|            42.717| 267532|
|            42.657| 779399|
|            42.648|1311154|
|42.638999999999996| 239546|
|             42.62| 623075|
|             42.65| 499117|
|            42.725|  95826|
|            42.724| 670740|
|            42.662| 719961|
|          42.702594|  31947|
|             42.58| 351810|
|            42.675|1416094|
|            42.651| 772961|
|            42.683|1258500|
| 42.61600000000001| 448188|
|42.736999999999995| 342613|
|42.611000000000004| 191893|
|            42.716| 472659|
|42.611999999999995| 236382|
|42.638000000000005|  35882|
+------------------+-------+
only showing top 20 rows
```

```
scala> df.where("pressure > 94000").where("temperature > 28").show()
21/12/04 10:51:09 WARN CSVHeaderChecker: CSV header does not conform to the schema.
 Header: , sensor_id, location, lat, lon, timestamp, pressure, temperature, humidity
 Schema: _c0, sensor_id, location, lat, lon, timestamp, pressure, temperature, humidity
Expected: _c0 but found:
CSV file: file:///home/ravi/BDM/Project/Sofia%20Air%20quality/2017-10_bme280sof.csv
+-----+---------+--------+-----------------+-----------------+-------------------+--------+-----------+--------+
|  _c0|sensor_id|location|              lat|              lon|          timestamp|pressure|temperature|humidity|
+-----+---------+--------+-----------------+-----------------+-------------------+--------+-----------+--------+
|62858|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:19:41|96064.89|      28.81|   24.89|
|63093|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:22:08|96071.48|      29.22|    24.8|
|63328|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:24:34|96075.89|      30.51|   22.65|
|63565|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:27:01|96075.84|      32.55|   20.16|
|63803|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:29:28| 96073.2|      34.49|   18.33|
|64039|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:31:55|96088.52|      35.85|   17.07|
|64272|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:34:21|96075.99|      36.65|   15.71|
|64507|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:36:48|96074.42|      37.13|   15.07|
|64736|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:39:14|96074.31|      37.05|   15.14|
|64962|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:41:41|96075.95|      36.21|   15.32|
|65186|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:44:08|96076.45|      35.26|   16.93|
|65412|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:46:34|96064.88|      35.17|   16.44|
|65638|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:49:01|96070.21|      35.55|   16.07|
|65868|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:51:27|96073.34|      35.97|   15.51|
|66093|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:53:54|96071.61|      36.28|   15.47|
|66320|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:56:20|96078.55|      36.73|   15.22|
|66540|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 11:58:47|96069.18|      36.88|   14.65|
|66763|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 12:01:13|96068.64|      36.68|   14.87|
|66983|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 12:03:40|96076.11|      35.15|   16.24|
|67204|     5470|    2757|42.733000000000004|23.293000000000003|2017-10-01 12:06:06|96068.37|      34.26|   16.88|
+-----+---------+--------+-----------------+-----------------+-------------------+--------+-----------+--------+
only showing top 20 rows
```

```
+-------+-------------+------------+------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|summary|          _c0|   sensor_id|    location|              lat|              lon|         pressure|      temperature|         humidity|
+-------+-------------+------------+------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|  count|     97288452|    97288452|    97288452|         97288452|         97288452|         97287604|         97288452|         97288231|
|   mean|9236315.971576195|8457.130418808596|4374.703005121307| 42.673445841908084|  23.32649411102254|  94739.30374934217|13.235649798394999| 63.13118144177176|
| stddev|7630335.900454502|4705.981840274353|2535.649673556141|0.03431472474319096|0.05069146053173863|  45350.03723028208|17.154591452469138|23.198811526037584|
|    min|            0|         740|         354| 42.571999999999996| 23.218000000000004|-3.38669502780000...|           -5572.8|          -9999.0|
|    max|     47642393|       29531|       16472| 42.778999999999996|           23.482|         210750.28|           192.05|            100.0|
+-------+-------------+------------+------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

Spark Streaming:

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val pressure_avg = data_streaming.select(avg("pressure")).writeStream.queryName("Average_pressure").format("console").outputMode("complete").start()
21/12/04 12:49:11 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-f7538c53-4483-4791-a648-9b70c82e0e19. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
pressure_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@7fcc9949

scala>

scala>

-------------------------------------------
Batch: 0
-------------------------------------------
+-----------------+
|    avg(pressure)|
+-----------------+
|94706.86233328514|
+-----------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------+
|    avg(pressure)|
+-----------------+
|94809.97640391887|
+-----------------+
```

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val temperature_avg = data_streaming.select(avg("temperature")).writeStream.queryName("Average_temperature").format("console").outputMode("complete").start()
21/12/04 13:41:04 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-988654bd-5491-4c2e-91ed-ac632d5639ce. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
temperature_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@534ffe23

-------------------------------------------
Batch: 0
-------------------------------------------
+------------------+
|  avg(temperature)|
+------------------+
|21.928538168466623|
+------------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+------------------+
| avg(temperature)|
+------------------+
|23.061539400837341|
+------------------+
```

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val humidity_avg = data_streaming.select(avg("humidity")).writeStream.queryName("Average_humidity").format("console").outputMode("complete").start()
21/12/04 13:38:39 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-a07aaaff-495b-49cf-b53f-cd9999d9600a. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
humidity_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@2b953930

-------------------------------------------
Batch: 0
-------------------------------------------
+----------------+
|   avg(humidity)|
+----------------+
|47.565439527536|
+----------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+------------------+
|    avg(humidity)|
+------------------+
|47.079514500480895|
+------------------+
```

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val time = data_streaming.withColumn("month",month(col("timestamp"))).withColumn("date",dayofmonth(col("timestamp"))).withColumn("year",year(col("timestamp")))
time: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 10 more fields]

scala> val year_avg = time.groupBy("year").avg("pressure","temperature","humidity").select(col("year"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_year").for
mat("console").outputMode("update").start()
21/12/04 13:42:42 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-f56e0e67-5b41-40f2-b88e-b23ac9896cb7. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
year_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@539edd3

-------------------------------------------
Batch: 0
-------------------------------------------
+----+-----------------+-----------------+----------------+
|year|    avg(pressure)|  avg(temperature)|   avg(humidity)|
+----+-----------------+-----------------+----------------+
|2017|94706.06233328514|21.928538168466623|47.5654395727536|
+----+-----------------+-----------------+----------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+----+-----------------+-----------------+----------------+
|year|    avg(pressure)| avg(temperature)|    avg(humidity)|
+----+-----------------+-----------------+----------------+
|2017|94809.97640391887|23.061539400373 41|47.87951450840895|
+----+-----------------+-----------------+----------------+
```

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val time = data_streaming.withColumn("month",month(col("timestamp"))).withColumn("date",dayofmonth(col("timestamp"))).withColumn("year",year(col("timestamp")))
time: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 10 more fields]

scala> val month_avg = time.groupBy("month").avg("pressure","temperature","humidity").select(col("month"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_month")
.format("console").outputMode("update").start()
21/12/04 13:44:32 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-6d468819-5b40-4848-b049-54afd4015de9. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
month_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@594d6d25

-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-----------------+-----------------+----------------+
|month|    avg(pressure)|  avg(temperature)|   avg(humidity)|
+-----+-----------------+-----------------+----------------+
|    9|94706.06233328514|21.928538168466623|47.5654395727536|
+-----+-----------------+-----------------+----------------+

-------------------------------------------
Batch: 1
-------------------------------------------
+-----+-----------------+-----------------+----------------+
|month|    avg(pressure)|  avg(temperature)|    avg(humidity)|
+-----+-----------------+-----------------+----------------+
|    7|94965.2925000588|24.754989836760743|48.34894942327553|
+-----+-----------------+-----------------+----------------+
```

```
scala> val data = spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")
data: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val dataSchema = data.schema
dataSchema: org.apache.spark.sql.types.StructType = StructType(StructField(_c0,IntegerType,true), StructField(sensor_id,IntegerType,true), StructField(location,IntegerType,true), StructField(lat,DoubleType,true)
, StructField(lon,DoubleType,true), StructField(timestamp,TimestampType,true), StructField(pressure,DoubleType,true), StructField(temperature,DoubleType,true), StructField(humidity,DoubleType,true))

scala> val data_streaming = spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality_/*")
data_streaming: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 7 more fields]

scala> val time = data_streaming.withColumn("month",month(col("timestamp"))).withColumn("date",dayofmonth(col("timestamp"))).withColumn("year",year(col("timestamp")))
time: org.apache.spark.sql.DataFrame = [_c0: int, sensor_id: int ... 10 more fields]

scala> val date_avg = time.groupBy("date").avg("pressure","temperature","humidity").select(col("date"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_date").for
mat("console").outputMode("update").start()
21/12/04 13:44:27 WARN StreamingQueryManager: Temporary checkpoint location created which is deleted normally when the query didn't fail: /tmp/temporary-b51ef849-4b6e-4d4e-8d02-ac92657c388c. If it's required to
delete it under any circumstances, please set spark.sql.streaming.forceDeleteTempCheckpointLocation to true. Important to know deleting temp checkpoint folder is best effort.
date_avg: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.streaming.StreamingQueryWrapper@5aea3e4e

-------------------------------------------
Batch: 0
-------------------------------------------
+----+-----------------+-----------------+-----------------+
|date|    avg(pressure)|  avg(temperature)|    avg(humidity)|
+----+-----------------+-----------------+-----------------+
|  26|95137.62740508349|15.584336913750988| 74.92852919281044|
|  12|94158.50115709259| 24.56104727341289| 45.89601333836076|
|  22|94976.00788629203|13.749735178389912| 59.48010212653099|
|   1|94683.60519262978|25.022768931601124| 43.39138686131375|
|  13|94984.05672285167|22.534218338520496| 47.52533445056276|
|   6|94938.54340308974|19.612906629554544|  43.603745445344|
|  16|94799.59217234189| 25.80753826175655| 41.43912925517127|
|   3|94169.16093847207|22.815941459525483|54.449244326755974|
|  20| 94212.2309710042|20.989666442595215|51.862045450504004|
|   5| 95140.0617798725|17.865410925726636|48.154461248654364|
|  19|94303.68676405793| 25.01795806451607|38.758865207373326|
|  15|94689.40534227456| 24.80976263257869|43.959721248580074|
|   9|94478.35230896981|22.400644396500503| 53.57615566988295|
|  17|94698.73303812215|27.301447725423056|36.661146106538666|
|   4|94894.57666902986| 18.83945800861075| 51.40765841550099|
|   8|94692.76728382864|23.866544022242886| 44.34606139944391|
|  23| 95248.8698113321|16.050415296687742|53.399415892489976|
|   7| 94787.7196167973|22.227425693055213| 42.43878989235004|
|  10|  94518.228190255|23.574929638242146|50.363449044662545|
|  25|95162.64129548067|18.488955707181663|51.432771597085996|
+----+-----------------+-----------------+-----------------+
only showing top 20 rows
```

Machine Learning:

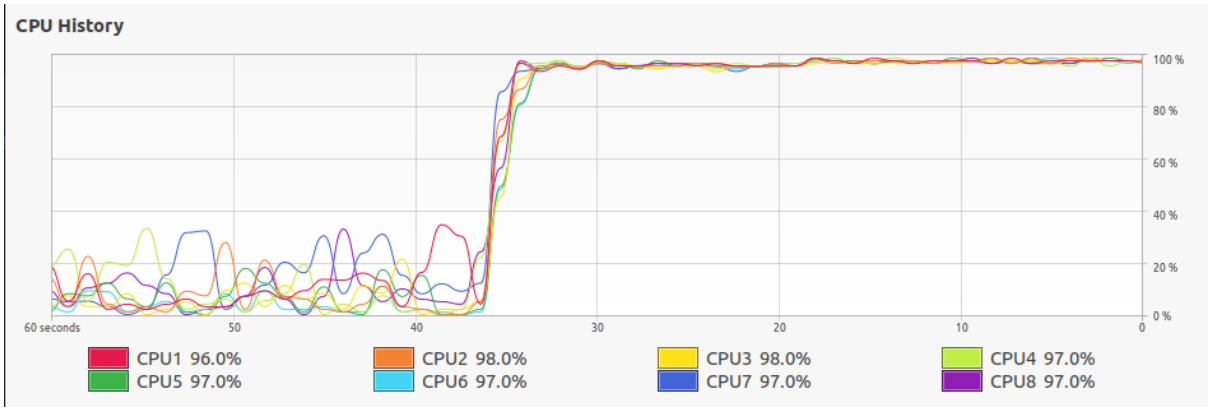
```
Using Python version 3.8.10 (default, Sep 28 2021 16:10:42)
Spark context Web UI available at http://192.168.0.105:4040
Spark context available as 'sc' (master = local[*], app id = local-1638606184096).
SparkSession available as 'spark'.
root
 |-- _c0: integer (nullable = true)
 |-- sensor_id: integer (nullable = true)
 |-- location: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- lon: double (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- pressure: double (nullable = true)
 |-- temperature: double (nullable = true)
 |-- humidity: double (nullable = true)

Row(_c0=11, sensor_id=3436, location=1731, lat=42.693, lon=23.359, timestamp=datetime.datetime(2017, 9, 1, 0, 0, 10), pressure=95313.38, temperature=20.1, humidity=47.61)
['_c0', 'sensor_id', 'location', 'lat', 'lon', 'timestamp', 'pressure', 'temperature', 'humidity']
21/12/04 13:53:12 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
21/12/04 13:53:12 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
+------------------+--------+---------------+
|        prediction|humidity|       features|
+------------------+--------+---------------+
| 43.94358955363248|    42.6|[91400.39,26.04]|
| 44.48404595765146|   42.92| [91405.2,25.77]|
| 44.05952979084401|   45.14|[91418.56,25.97]|
|43.713358236475216|   38.99|[91419.65,26.14]|
|  45.4155151634923|   40.11| [91420.1,25.3]|
+------------------+--------+---------------+
only showing top 5 rows

Root Mean Squared Error (RMSE) on test data for Linear Regression = 7.63355
R Squared (R2) on test data for  Linear Regression = 0.762055
+-----------------+--------+---------------+
|       prediction|humidity|       features|
+-----------------+--------+---------------+
|38.25905373358782|    42.6|[91400.39,26.04]|
|38.25905373358782|   42.92| [91405.2,25.77]|
|38.25905373358782|   45.14|[91418.56,25.97]|
|38.25905373358782|   38.99|[91419.65,26.14]|
|41.63008214496943|   40.11| [91420.1,25.3]|
+-----------------+--------+---------------+
only showing top 5 rows

rootMeanSquaredError (rmse) on test data for Decision Tree Algorithm = 7.25827
R Squared (R2) on test data for Decision Tree Algorithm = 0.784875
```

Parallel Processing:




```
scala> sc.defaultParallelism
res10: Int = 8
```

CONCLUSION

A Spark analysis and data prediction system is being designed where Analysis of data is done using DataFrames and streaming; Two ML Models i.e., one with Linear Regression and the other with Decision Tree Algorithm are built using Spark ML Library, which predicted humidity with accuracies 0.76 and 0.78. Decision Tree is a bit more suitable for this type of data.

REFERENCES

1. High-performance IoT streaming data prediction system using Spark: a case study of air pollution Ho-Yong-Jin , Eun-Sung Jung, Duckki Lee
2. https://towardsdatascience.com/building-a-linear-regression-with-pyspark-and-mllib-d065c3ba246a
3. https://www.youtube.com/watch?v=YKP31T5LIXQ&t=153s

ANNEXURE

Spark DataFrames Queries:

```
val df =
spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/Sofia Air quality/*")

df.columns

df.show(false)

df.count

df.printSchema()

df.groupBy("sensor_id").count().show()
df.groupBy("location").count().show()
df.groupBy("lon").count().show()
df.groupBy("lat").count().show()

df.describe().show()

import org.apache.spark.sql.functions.{min, max}
df.select(min("pressure"), max("pressure")).show()
df.select(min("temperature"), max("temperature")).show()
df.select(min("humidity"), max("humidity")).show()


df.where("pressure > 94000").where("temperature > 28").show()
df.where("pressure > 94000").where("temperature > 28").count()
```

Structured Streaming Queries:

```
val data =
spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/ravi/BDM/Project/test.csv")

val dataSchema = data.schema

val data_streaming =
spark.readStream.option("header","true").schema(dataSchema).option("maxFilesPerTrigger", 1).csv("/home/ravi/BDM/Project/Sofia Air quality/*")

val pressure_avg =
data_streaming.select(avg("pressure")).writeStream.queryName("Average_pressure").format("console").outputMode("complete").start()

val humidity_avg =
data_streaming.select(avg("humidity")).writeStream.queryName("Average_humidity").format("console").outputMode("complete").start()

val temperature_avg =
data_streaming.select(avg("temperature")).writeStream.queryName("Average_temperature").format("console").outputMode("complete").start()

val time =
data_streaming.withColumn("month",month(col("timestamp"))).withColumn("date",dayofmonth(col("timestamp"))).withColumn("year",year(col("timestamp")))

val year_avg =
time.groupBy("year").avg("pressure","temperature","humidity").select(col("year"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_year").format("console").outputMode("update").start()

val month_avg =
time.groupBy("month").avg("pressure","temperature","humidity").select(col("month"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_month").format("console").outputMode("update").start()

val date_avg =
time.groupBy("date").avg("pressure","temperature","humidity").select(col("date"),col("avg(pressure)"),col("avg(temperature)"),col("avg(humidity)")).writeStream.queryName("Average_date").format("console").outputMode("update").start()
```

Parallel Processing Queries:

```
sc.defaultParallelism
```

Building ML Model:

```
from pyspark.sql import SparkSession
spark=SparkSession.builder.appName("lin_reg").getOrCreate()
df =
spark.read.format("csv").option("header","true").option("inferSchema","true").load("/home/suryateja/spark_files/data/Sofia Air quality/2017-09_bme280sof.csv")
df.printSchema()
print(df.head())
df =df.na.drop()

from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

print(df.columns)

assembler=VectorAssembler(inputCols=['pressure', 'temperature'],
                          outputCol='features')

output=assembler.transform(df)

final_df=output.select('features','humidity')
train_data,test_data= final_df.randomSplit([0.7,0.3])

from pyspark.ml.regression import LinearRegression

lm=LinearRegression(featuresCol='features',labelCol='humidity',maxIter=10,regParam=0.3,elasticNetParam=0.8)
model=lm.fit(train_data)
pred=model.transform(test_data)
pred.select("prediction","humidity","features").show(5)
from pyspark.ml.evaluation import RegressionEvaluator

res=model.evaluate(test_data)
print("Root Mean Squared Error (RMSE) on test data for Linear Regression = %g" % res.rootMeanSquaredError)
print("R Squared (R2) on test data for  Linear Regression = %g" % res.r2)

from pyspark.ml.regression import DecisionTreeRegressor
dt = DecisionTreeRegressor(featuresCol ='features', labelCol = 'humidity')
dt_model = dt.fit(train_data)
```

```
dt_predictions = dt_model.transform(test_data)

dt_predictions.select("prediction","humidity","features").show(5)

dt_rmse_evaluator = RegressionEvaluator(labelCol="humidity", predictionCol="prediction",
metricName="rmse")
rmse = dt_rmse_evaluator.evaluate(dt_predictions)
print("rootMeanSquaredError (rmse) on test data for Decision Tree Algorithm = %g" %
rmse)

dt_r2_evaluator = RegressionEvaluator(labelCol="humidity", predictionCol="prediction",
metricName="r2")
r2 = dt_r2_evaluator.evaluate(dt_predictions)
print("R Squared (R2) on test data for Decision Tree Algorithm = %g" % r2)
```