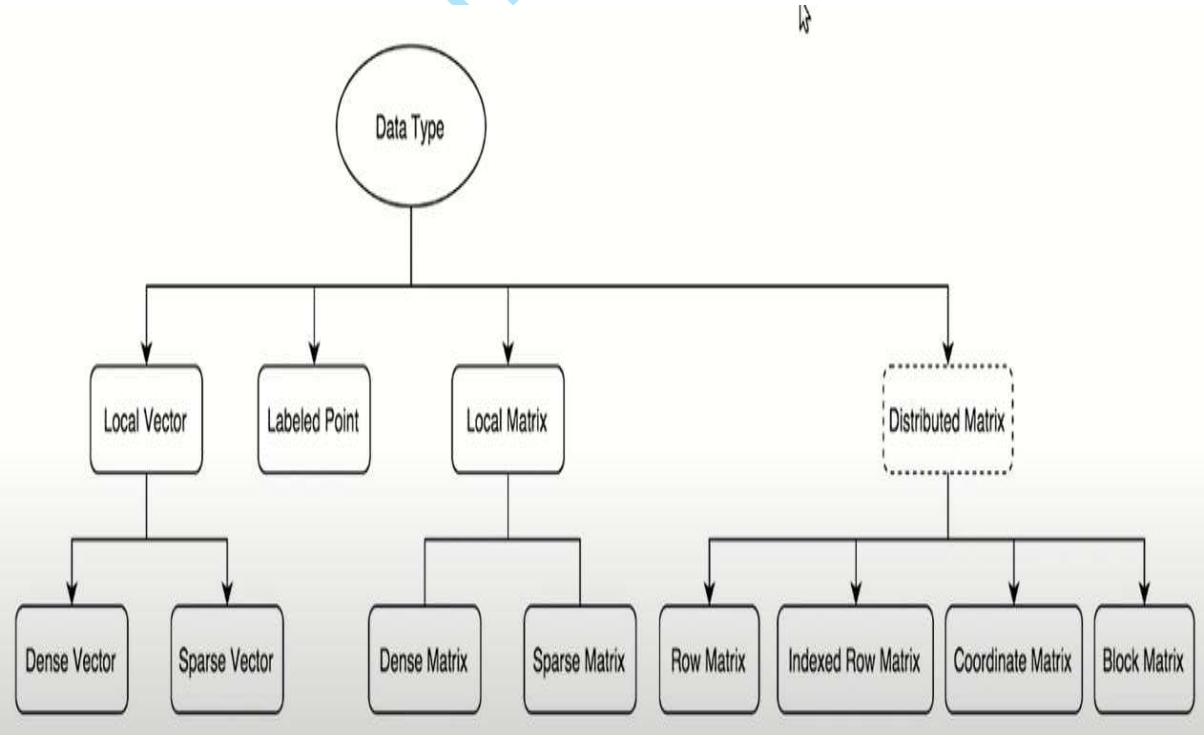
Introduction to MLlib

Introduction

- MLlib is a cool tool within Apache Spark that helps us do fancy stuff with data using machine learning. Think of it like a big toolbox full of algorithms and tools to help us analyze and make predictions with our data. Whether we want to classify emails, recommend movies, or predict prices, MLlib's got our back. It's like having a super-smart assistant for our data analysis tasks!
- It provides tools to implement all machine learning algorithms, including Regression, classification, dimensionality reduction tools, transformation, feature extraction, pipelines (tunning), save and load algorithm, and utilities for linear algebra and statistics.

Understanding Basic Data Types



Sparse and dense are two different ways to represent data, particularly vectors and matrices.

Sparse	**Dense**
• In a dense representation, **all elements are explicitly stored**, regardless of whether they are zero or non-zero. • For example, if we have a vector [1.0, 0.0, 3.0], in a dense representation, we would store all three elements [1.0, 0.0, 3.0] even though one element is zero. • Dense representations are straightforward and easy to work with, but they can be memory-intensive, especially for large datasets with many zero values.	• In a sparse representation, **only non-zero elements are explicitly stored**, along with their indices. • For the same vector [1.0, 0.0, 3.0], in a sparse representation, we would only store the non-zero elements [1.0, 3.0] along with their corresponding indices [0, 2]. • Sparse representations are memory-efficient, especially for datasets where most values are zero, but they may require special handling in algorithms.

MLlib in Apache Spark supports several basic data types for machine learning tasks. Here are a few examples:

Vectors	from pyspark.ml.linalg import Vectors dense_vector = Vectors.dense([1.0, 2.0, 3.0]) sparse_vector = Vectors.sparse(3, [0, 2], [1.0, 3.0])

	In the context of sparse representation, `[0, 2]` represents the indices of the non-zero elements, and `[1.0, 3.0]` represents the corresponding non-zero values. So, if we interpret this in terms of a vector: • `[0, 2]` indicates that the non-zero elements are located at indices 0 and 2. • `[1.0, 3.0]` indicates the values at these indices, which are `1.0` at index 0 and `3.0` at index 2. This means that the sparse vector being represented is `[1.0, 0.0, 3.0]`, but instead of storing all three elements, we only store the non-zero elements (`1.0` and `3.0`) along with their respective indices (`0` and `2`). This is a memory-efficient way of representing vectors, especially when the majority of elements are zero.
Labeled Point	Labeled points represent labeled instances in supervised learning. Each labeled point consists of a label and a feature vector. In supervised learning, we typically deal with labeled data. Labeled data consists of examples where each example has both input features (also known as attributes or independent variables) and an associated output label (also known as the target or dependent variable). 1. **Feature Vector**: This represents the input features of an example. It's essentially an array or a vector where each element corresponds to a specific feature. For example, if we're trying to predict housing prices, the features might include things like the number of bedrooms, square footage, and neighborhood crime rate. Each of these features would be a component of the feature vector. 2. **Label**: This is the output value that we're trying to predict or classify. In regression problems, the label is typically a continuous value, such as the price of a house. In classification problems, the label is a categorical value representing the class or category to which the example belongs, such as "spam" or "not spam" for email classification. Now, a labeled point combines these two components into a single entity: • **Labeled Point**: It's a data structure that pairs a label (the output value) with a feature vector (the input features). This pairing represents one example or instance in the dataset. For example, in a dataset for predicting housing prices, a labeled point might consist of the price of a house (the label) paired with its corresponding features like the number of bedrooms, square footage, and crime rate in the neighborhood (the feature vector).

	python Copy code
	`from pyspark.mllib.regression import LabeledPoint`
	`# Example labeled point for predicting housing prices`
	`# Features: [num_bedrooms, square_footage, crime_rate]`
	`# Label: price`
	`labeled_point = LabeledPoint(250000, [3, 2000, 0.05])`
	Here, `250000` is the label (price of the house), and `[3, 2000, 0.05]` is the feature vector representing the number of bedrooms, square footage, and crime rate, respectively. This labeled point represents one example in our dataset, where we have the features of a house and its corresponding price.
	For example:
	from pyspark.mllib.regression import LabeledPoint
	labeled_point = LabeledPoint(1.0, [0.0, 1.0, 2.0])

Local Matrix	Matrices represent two-dimensional arrays of numbers. They can be dense or sparse.
	1. **Dense Matrix:**
	• `Matrices.dense(2, 2, [1.0, 2.0, 3.0, 4.0])`: This line creates a dense matrix.
	• `2, 2`: Specifies the dimensions of the matrix, in this case, it's a 2×2 matrix.
	• `[1.0, 2.0, 3.0, 4.0]`: This list `[1.0, 2.0, 3.0, 4.0]` represents the elements of the matrix. It's stored in row-major order, meaning the elements are listed row by row.
	• So, the resulting dense matrix would look like:
	csharp Copy code
	`[1.0, 2.0]`
	`[3.0, 4.0]`

	2. **Sparse Matrix**: • `Matrices.sparse(2, 2, [0, 1], [0, 1], [1.0, 2.0])`: This line creates a sparse matrix. • `2, 2`: Specifies the dimensions of the matrix, still a 2×2 matrix. • `[0, 1], [0, 1]`: These two lists `[0, 1]` represent the row indices and `[0, 1]` represent the column indices of the non-zero elements. • `[1.0, 2.0]`: This list `[1.0, 2.0]` represents the non-zero elements of the matrix. • So, the resulting sparse matrix would look like: csharp Copy code [1.0, 0.0] [0.0, 2.0] In the sparse matrix, the non-zero elements are `[1.0, 2.0]` located at the positions `(0, 0)` and `(1, 1)` respectively, while the zero elements are not stored explicitly, making it a more memory-efficient representation compared to the dense matrix. from pyspark.ml.linalg import Matrices dense_matrix = Matrices.dense(2, 2, [1.0, 2.0, 3.0, 4.0]) sparse_matrix = Matrices.sparse(2, 2, [0, 1], [0, 1], [1.0, 2.0])
Distributed Matrix	A distributed matrix in Apache Spark is a data structure used to represent large matrices that are partitioned across multiple nodes in a cluster. This allows Spark to efficiently process and analyze large-scale data that cannot fit into the memory of a single machine. 1. **Row Matrix**: • A row matrix is a distributed matrix where each row is treated as a separate vector. • It is suitable for representing matrices where the number of rows is relatively small compared to the number of columns. • Row matrices are typically used for simple matrix operations and are most efficient when the number of rows fits comfortably in memory across the cluster.

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import RowMatrix

# Create some sample data
data = [
    Vectors.dense([1.0, 2.0, 3.0]),
    Vectors.dense([4.0, 5.0, 6.0]),
    Vectors.dense([7.0, 8.0, 9.0])
]

# Create a RowMatrix from the data
row_matrix = RowMatrix(sc.parallelize(data))

# Perform operations on the row matrix
row_matrix.rows.collect()
```

2. **Indexed Row Matrix:**

- An indexed row matrix is a distributed matrix similar to a row matrix, but it also associates each row with a unique index or key.
- This index allows for efficient row-based lookup and manipulation operations.
- Indexed row matrices are useful when you need to perform operations that involve accessing or updating individual rows based on their index.

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.linalg.distributed import IndexedRow, IndexedRowMatrix

# Create some sample data
data = [
```

```
    IndexedRow(0, Vectors.dense([1.0, 2.0, 3.0])),
    IndexedRow(1, Vectors.dense([4.0, 5.0, 6.0])),
    IndexedRow(2, Vectors.dense([7.0, 8.0, 9.0]))
]

# Create an IndexedRowMatrix from the data
indexed_row_matrix = IndexedRowMatrix(sc.parallelize(data))

# Perform operations on the indexed row matrix
indexed_row_matrix.rows.collect()
```

3. **Coordinate Matrix**:
   - A coordinate matrix is a distributed matrix represented as a collection of tuples `(i, j, value)`, where `i` and `j` are the row and column indices, and `value` is the matrix element.
   - It is well-suited for sparse matrices where the majority of elements are zero.
   - Coordinate matrices are efficient for constructing and manipulating sparse matrices, especially when the matrix dimensions are large and the number of non-zero elements is relatively small.

```
from pyspark.mllib.linalg.distributed import CoordinateMatrix

# Create some sample data
data = [
    (0, 0, 1.0),
    (0, 2, 3.0),
    (1, 1, 4.0),
    (2, 0, 7.0),
    (2, 2, 9.0)
]
```

```
# Create a CoordinateMatrix from the data
coordinate_matrix = CoordinateMatrix(sc.parallelize(data))

# Perform operations on the coordinate matrix
coordinate_matrix.entries.collect()
```

4. **Block Matrix:**
   - A block matrix is a distributed matrix partitioned into fixed-size blocks, where each block is a submatrix.
   - It is designed for efficient handling of large-scale matrices by partitioning them into smaller, manageable blocks.
   - Block matrices are particularly useful for distributed linear algebra operations, such as matrix multiplication and inversion, as they exploit the inherent parallelism in block-wise operations.

```
from pyspark.mllib.linalg.distributed import BlockMatrix

# Create some sample data
rows = [
    [1.0, 2.0, 0.0],
    [0.0, 3.0, 4.0],
    [5.0, 0.0, 6.0]
]

# Create a BlockMatrix from the data
block_matrix = BlockMatrix(sc.parallelize(rows), rowsPerBlock=2, colsPerBlock=2)

# Perform operations on the block matrix
```

	block_matrix.blocks.collect() In the example provided, `**rowsPerBlock=2**` and `**colsPerBlock=2**` are parameters used when creating a `**BlockMatrix**`. These parameters define how the original matrix is partitioned into blocks. • `**rowsPerBlock**`: Specifies the number of rows in each block of the resulting block matrix. • `**colsPerBlock**`: Specifies the number of columns in each block of the resulting block matrix. For example, if you have a matrix with 4 rows and 4 columns, and you set `**rowsPerBlock=2**` and `**colsPerBlock=2**`, the matrix will be partitioned into 4 blocks, each with 2 rows and 2 columns. This partitioning is helpful for distributed computation because it allows Spark to process different parts of the matrix in parallel across multiple nodes in the cluster. It also helps in optimizing memory usage and computation efficiency.

Linear Regression Model Boston Housing Dataset

In [14]:
```
from pyspark.sql import SparkSession
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator
```

In [15]:
```
spark = SparkSession.builder \
    .appName("Boston Housing Linear Regression") \
    .getOrCreate()
```

In [16]:
```
spark
```

Out[16]:

SparkSession - in-memory

SparkContext

Spark UI (http://DESKTOP-OCOSJOH.bbrouter:4041)

Version

v3.5.0

Master

local[*]

AppName

Boston Housing Linear Regression

In [17]:
```
file_path = "C:\\Spark\\sparkfiles\\boston_housing 1.csv"
df = spark.read.csv(file_path, header=True, inferSchema=True)
```

In [18]:
```
df.show()
```

```
+-------+----+-----+----+-----+-----+-----+------+---+---+-------+------+-----+----+
|   crim|  zn|indus|chas|  nox|   rm|  age|   dis|rad|tax|ptratio|     b|lstat|medv|
+-------+----+-----+----+-----+-----+-----+------+---+---+-------+------+-----+----+
|0.00632|18.0| 2.31|   0|0.538|6.575| 65.2|  4.09|  1|296|   15.3| 396.9| 4.98|24.0|
|0.02731| 0.0| 7.07|   0|0.469|6.421| 78.9|4.9671|  2|242|   17.8| 396.9| 9.14|21.6|
|0.02729| 0.0| 7.07|   0|0.469|7.185| 61.1|4.9671|  2|242|   17.8|392.83| 4.03|34.7|
|0.03237| 0.0| 2.18|   0|0.458|6.998| 45.8|6.0622|  3|222|   18.7|394.63| 2.94|33.4|
|0.06905| 0.0| 2.18|   0|0.458|7.147| 54.2|6.0622|  3|222|   18.7| 396.9| 5.33|36.2|
|0.02985| 0.0| 2.18|   0|0.458| 6.43| 58.7|6.0622|  3|222|   18.7|394.12| 5.21|28.7|
|0.08829|12.5| 7.87|   0|0.524|6.012| 66.6|5.5605|  5|311|   15.2| 395.6|12.43|22.9|
|0.14455|12.5| 7.87|   0|0.524|6.172| 96.1|5.9505|  5|311|   15.2| 396.9|19.15|27.1|
|0.21124|12.5| 7.87|   0|0.524|5.631|100.0|6.0821|  5|311|   15.2|386.63|29.93|16.5|
|0.17004|12.5| 7.87|   0|0.524|6.004| 85.9|6.5921|  5|311|   15.2|386.71| 17.1|18.9|
|0.22489|12.5| 7.87|   0|0.524|6.377| 94.3|6.3467|  5|311|   15.2|392.52|20.45|15.0|
|0.11747|12.5| 7.87|   0|0.524|6.009| 82.9|6.2267|  5|311|   15.2| 396.9|13.27|18.9|
|0.09378|12.5| 7.87|   0|0.524|5.889| 39.0|5.4509|  5|311|   15.2| 390.5|15.71|21.7|
|0.62976| 0.0| 8.14|   0|0.538|5.949| 61.8|4.7075|  4|307|   21.0| 396.9| 8.26|20.4|
|0.63796| 0.0| 8.14|   0|0.538|6.096| 84.5|4.4619|  4|307|   21.0|380.02|10.26|18.2|
|0.62739| 0.0| 8.14|   0|0.538|5.834| 56.5|4.4986|  4|307|   21.0|395.62| 8.47|19.9|
|1.05393| 0.0| 8.14|   0|0.538|5.935| 29.3|4.4986|  4|307|   21.0|386.85| 6.58|23.1|
| 0.7842| 0.0| 8.14|   0|0.538| 5.99| 81.7|4.2579|  4|307|   21.0|386.75|14.67|17.5|
|0.80271| 0.0| 8.14|   0|0.538|5.456| 36.6|3.7965|  4|307|   21.0|288.99|11.69|20.2|
| 0.7258| 0.0| 8.14|   0|0.538|5.727| 69.5|3.7965|  4|307|   21.0|390.95|11.28|18.2|
+-------+----+-----+----+-----+-----+-----+------+---+---+-------+------+-----+----+
only showing top 20 rows
```


In [19]:
```
feature_columns = df.columns[:-1]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
df_assembled = assembler.transform(df)
```

In [20]:
```
(train_data, test_data) = df_assembled.randomSplit([0.8, 0.2], seed=42)
```

In [21]:
```
# Train the linear regression model
lr = LinearRegression(labelCol="medv", featuresCol="features")
lr_model = lr.fit(train_data)
```

In [22]:
```
# Make predictions
predictions = lr_model.transform(test_data)
```

In [23]:
```
evaluator = RegressionEvaluator(labelCol="medv", predictionCol="prediction", metricName="rms
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = {:.2f}".format(rmse))
```

```
Root Mean Squared Error (RMSE) on test data = 4.67
```

In [24]:
```
spark.stop()
```

In [ ]:
```

```