Spark ML

Table of Content

Intro to Spark ML

Spark MLlib

- Spark MLlib is Apache Spark's *M*achine *L*earning *lib*rary
- It consists of algorithms like:
  - Classification
  - Regression
  - Clustering
  - Dimensionality Reduction
  - Collaborative Filtering

Spark MLlib

Algorithms

- Regression
- Classification
- Clustering
- Collaborative Filtering

Featurization

- Feature Extraction
- Feature Selection
- Transformation
- Dimensionality Reduction

Utilities

- Linear Algebra
- Statistics
- Data Handling

Pipeline

- Pipeline Construction
- Model Tuning
- Model Persistence

Spark MLlib and ML

- There are two machine learning implementations in Spark (ML and MLlib):
  - *spark.mllib* :- ML package built on top of the RDD API
  - *spark.ml* :- ML package built on top of higher-level DataFrame API

- Using spark.ml is recommended because the DataFrame API is more versatile and flexible

“Spark ML” is not an official name but used to refer to the MLlib DataFrame-based API (spark.ml)

Spark ML Pipeline

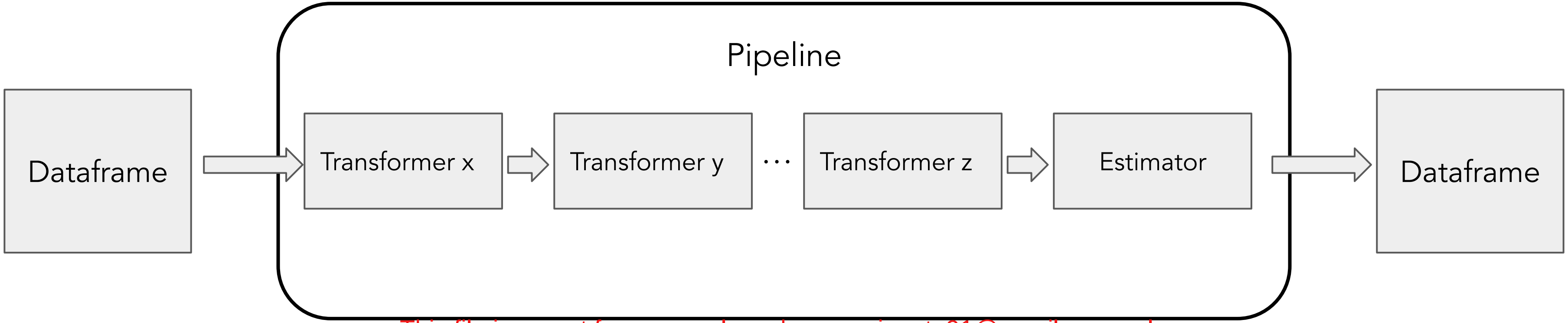
Spark ML Pipeline

- In machine learning, there are a lot of transformation steps that are performed to pre-process the data
- We repeat the same steps while making prediction
- You may often get confused about this transformations while working on huge projects
- To avoid this, pipelines were introduced that hold every step that is performed to fit the data on a model

Spark ML Pipeline

- The Pipeline API in Spark chains multiple Transformers and Estimator specifying a ML workflow

- It is a high-level API for MLlib that lives under the spark.ml package



Transformers

- A *Transformer* takes a dataset as input and produces an augmented dataset as output

- It basically *transforms* one DataFrame into another DataFrame

Estimators

- An *Estimator* fit on the input data that produces a model

- For eg., logistic regression is an Estimator that trains on a dataset with labels and features and produces a logistic regression model

- The model acts as a Transformer that transforms the input dataset

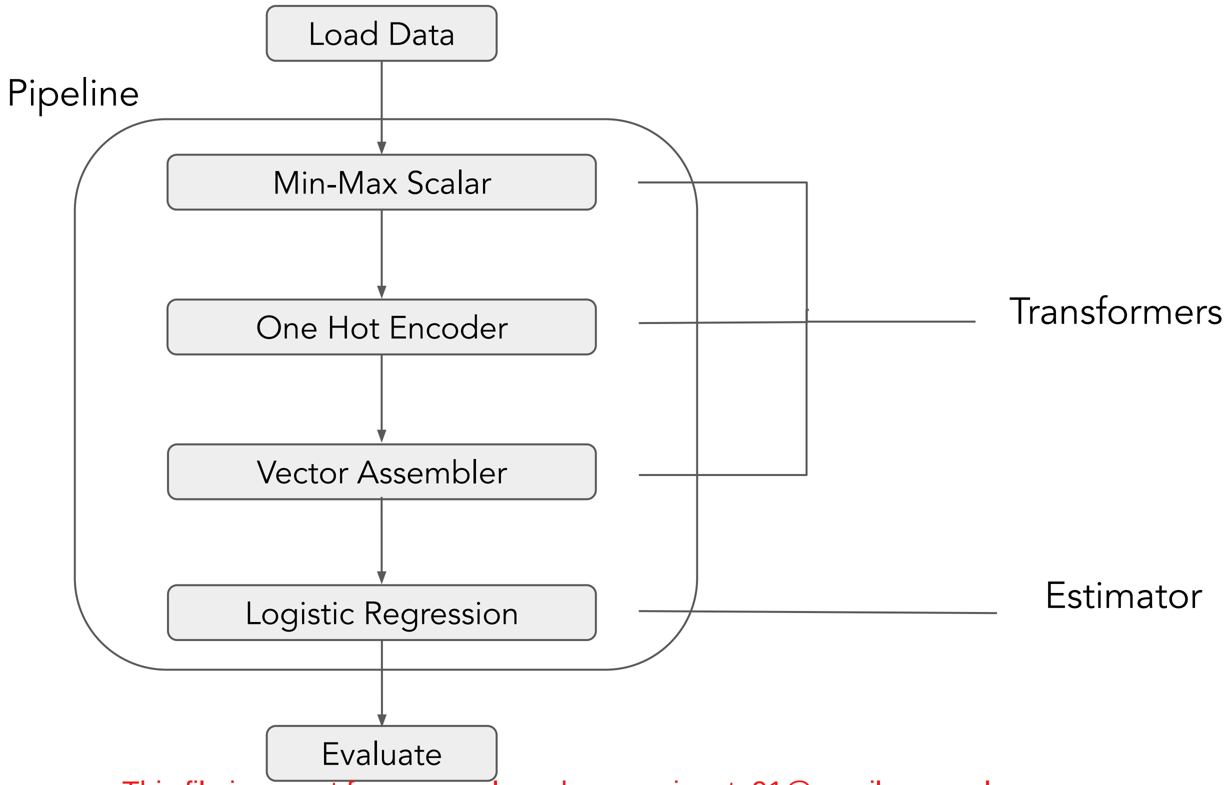
  For eg., logistic regression model can later be used to make predictions which technically adds prediction columns (Transformation) in the dataset

Spark ML Component Flow

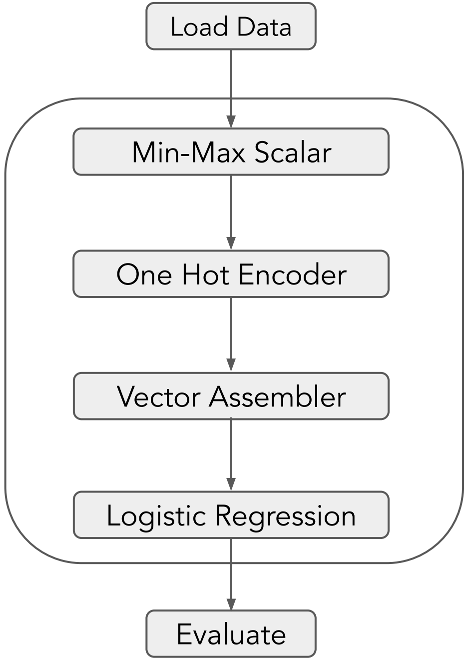
Spark ML Component Flow

- **Pipeline API** chains Transformers and Estimator each as a stage to specifying ML workflow
- These stages are run in order
- The input DataFrame is transformed as it passes through each stage
- **Evaluator** then evaluates the model performance

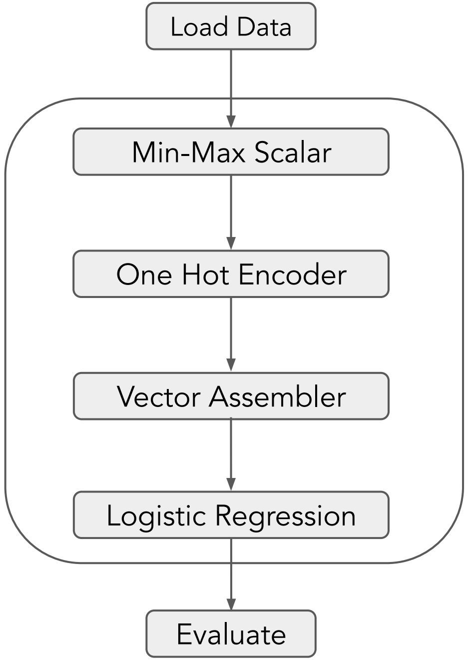
Spark ML Component Flow



Spark ML Component Flow



Pipeline of machine learning components



Reusing the pipeline on new data

Spark ML Component Flow

- Spark ML algorithms (estimators) expects all features to be contained within a single column in the form of Vector

- It is one of the important spark ml data types that you need to understand before we take a look at different feature transformers

Spark ML Data Types

Spark ML Data Types

- Spark ML uses the following data types internally for machine learning algorithms
  - Vectors
  - Matrix

Spark ML Data Types

- These data types help you for a process called featurization

- Conversion of numerical value, string value, character value, categorical value into numerical feature is called featurization

- The data once converted to these data types can be further passed to the ML algorithm in Spark

Spark ML Data Types

- For eg.: Consider the following sentences
  - I love programming
  - Python is a programming language
  - Python is my favourite programming language
  - Data science using Python

Spark ML Data Types

- Make list of the word such that one word should be occurring only once, then the list looks like as follow:

  [“I”, “love”, “programming”, “Python”, “is”, “a”, “language”, “my”, “favourite”, “Data”, “Science”, “using”]

- Now count occurrence of word in a sentence with respect to this list

Spark ML Data Types

- For example- vector conversion of sentence *"Data science using Python"* can be represented as :
  "I" - 0
  "love"- 0
  "programming" - 0
  "Python" - 1
  "is" - 0
  "a" - 0
  "language" - 0
  "my" - 0
  "favourite" - 0
  "Data" - 1
  "Science" - 1
  "using" - 1

Spark ML Data Types

- By following same approach other vector value are as follow:

  I love programming = [1 1 1 0 0 0 0 0 0 0 0 0]

  Python is a programming language = [0 0 1 1 1 1 1 0 0 0 0 0]

  Python is my favourite programming language = [0 0 1 1 1 0 1 1 1 0 0 0]

  Data science using Python = [0 0 0 1 0 0 0 0 0 1 1 1]

Spark ML Data Types

- And the sentences can also be converted into 4*12 matrix

```
array([[1 1 1 0 0 0 0 0 0 0 0 0],
       [0 0 1 1 1 1 1 0 0 0 0 0],
       [0 0 1 1 1 0 1 1 1 0 0 0],
       [0 0 0 1 0 0 0 0 0 1 1 1]])
```

- Now that you have understood vectors and matrix, let us now focus on the different types of vectors

Please Note

The elements of vectors and matrix are NOT always 0s and 1s.

Spark ML Data Types - Local Vector

- A local vector has integer-typed and 0-based indices and double-typed values
- They are stored on local machine
- A local vector can be represented as:
    - Dense Vector
    - Sparse Vector

Please Note

Dense and Sparse Vectors are vector representation of data

Spark ML Data Types - Local Vector

- Dense Vectors:
  - By definition, dense means closely compacted
  - Dense Vector is a vector representation that contains many values or values that are not zeros (very few zero values)
  - It is basically an array of values
  - For eg. A vector (3.0, 5.0, 8.0, 0.0) can be represented in dense format as [3.0, 5.0, 8.0, 0.0]

Spark ML Data Types - Local Vector

- Dense Vectors PySpark

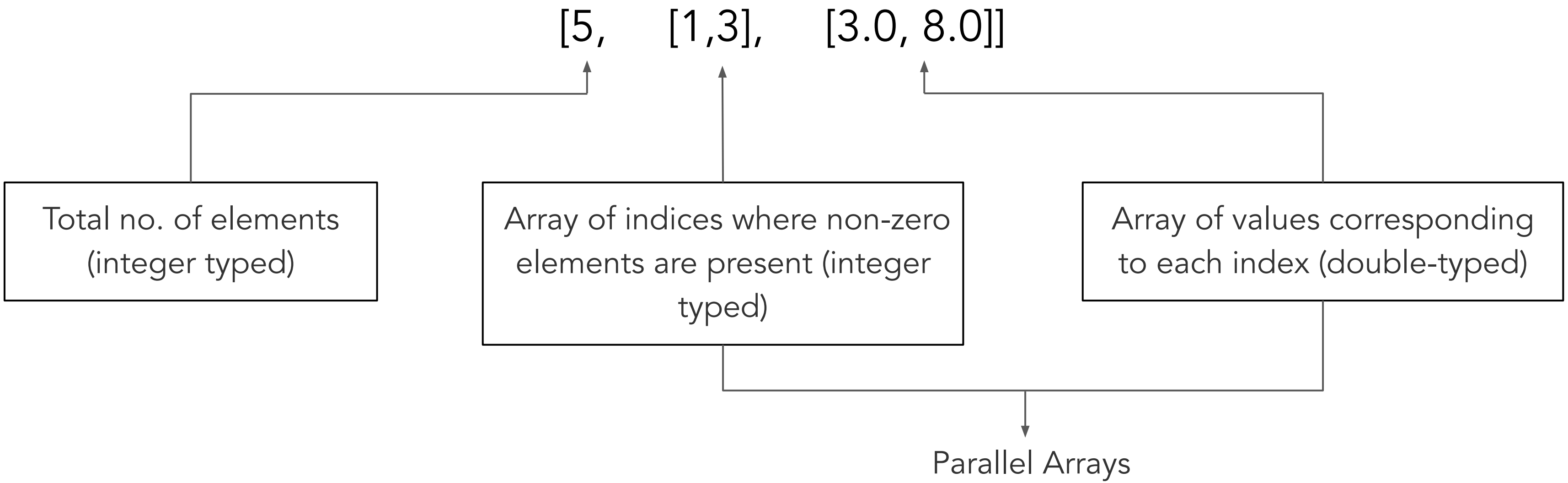
```
from pyspark.ml.linalg import Vectors
# Create a dense vector representation for (3.0, 5.0, 8.0, 0.0)
denseVector = Vectors.dense(3.0, 5.0, 8.0, 0.0)
denseVector
```

```
Out[1]: DenseVector([3.0, 5.0, 8.0, 0.0])
```

Spark ML Data Types - Local Vector

- Sparse Vectors:
  - By definition, sparse means thinly dispersed or scattered
  - If a vector has a majority of its elements as zero, it can be represented as sparse vector
  - It stores the size of the vector, an array of indices, and an array of values corresponding to those indices
  - For ex. A vector (0.0, 3.0, 0.0, 8.0, 0.0) can be represented in sparse format as [5, [1,3], [3.0, 8.0]]

Spark ML Data Types - Local Vector



- A sparse vector is used for storing non-zero entries for saving space

Spark ML Data Types - Local Vector

- Sparse Vector PySpark

```
from pyspark.ml.linalg import Vectors
# Create a sparse vector representation of (0.0, 3.0, 0.0, 8.0, 0.0)
sparseVector = Vectors.sparse(5, [1,3], [3.0, 8.0])
sparseVector
```

```
Out[2]: SparseVector(5, {1: 3.0, 3: 8.0})
```

In computer science, Parallel Array is an implicit data structure that contains multiple arrays

Each of these arrays are of the same size and the array elements are related to each other

i-th element of each array is closely related and all i-th elements together represent an object or entity

Spark ML Data Types - Labeled Point

- Labeled Point is a type of local vector
- It can either be dense or sparse
- It is associated with label/response variable
- Used in supervised learning algorithms
- A label should either be 0 (-ve) or 1 (+ve) for binary classification
- A label should be class indices starting from zero: 0, 1, 2,…

Spark ML Data Types - Labeled Point

Type	Label Values
Regression	Decimal Values
Binary Classification	0 or 1
Multi-class Classification	0,1,2,3, ….

Spark ML Transformers

ML Feature Transformers

- Feature building is a super important step for model building
- Some of the common feature transformer that we use for model building are:
  - Binarizer
  - Bucketizer
  - StringIndexer
  - IndexToString
  - OneHotEncoder
  - VectorAssembler
  - VectorIndexer
  - StandardScaler
  - MinMaxScaler
- Most Transforms are under *org.apache.spark.ml.feature* package

ML Feature Transformers

- Binarizer
  - Binarization is used for thresholding numerical feature to binary feature (0 or 1)
  - Binarizer takes inputCol, outputCol and threshold for binarization as parameter
  - Values greater than the threshold value are binarized to 1.0
  - Values less than the threshold value are binarized to 0.0

ML Feature Transformers

- Consider the following dataframe:

gender	age	diabetes	hypertension	stroke	heart disease	smoking history	BMI
Female	80.0	0	0	0	Yes	never	25.19
Female	36.0	0	0	0	No	current	23.45
Male	76.0	0	1	0	Yes	current	20.14
Female	44.0	1	0	0	No	never	19.31
Male	42.0	0	0	0	No	never	33.64

We can create a new variable "BodyType" by binarizing the 'BMI' variable (1- obese and 0- healthy) If your BMI is 30.0 or higher, the BodyType falls in the obese range)

ML Feature Transformers

- Code:

```
from pyspark.ml.feature import Binarizer
binarizer = Binarizer(inputCol="BMI", outputCol="BodyType", threshold=30.0)
binarizedDF = binarizer.transform(strokeDF)
binarizedDF.select('BMI', 'BodyType').show(5,False)
```

```
+-----+--------+
|BMI  |BodyType|
+-----+--------+
|25.19|0.0     |
|23.45|0.0     |
|20.14|0.0     |
|19.31|0.0     |
|33.64|1.0     |
+-----+--------+
```

Value equal to or above the threshold value 30 is set to 1 (denoting obesity) in the new column 'BodyType'

ML Feature Transformers

- Bucketizer
  - Bucketization is used for creating group of values of a continuous feature
  - Bucketizer takes inputCol, outputCol and splits for mapping continuous features into buckets as parameter
  - There are n buckets for n+1 splits
  - The splits that you provided have to be in strictly increasing order, i.e. $s0 < s1 < s2 < \ldots < sn$

ML Feature Transformers

- Code:

```
from pyspark.ml.feature import Bucketizer
# lets define the age age group splits
splits = [0, 25.0, 50.0, 75.0, 100.0]
bucketizer = Bucketizer(inputCol="age", outputCol="ageGroup", splits=splits)
bucketizedDF = bucketizer.transform(binarizedDF)
bucketizedDF.select('age', 'ageGroup').show(10,False)
```

```
+----+--------+
|age |ageGroup|
+----+--------+
|80.0|3.0     |
|36.0|1.0     |
|76.0|3.0     |
|44.0|1.0     |
|42.0|1.0     |
|54.0|2.0     |
|78.0|3.0     |
|67.0|2.0     |
|15.0|0.0     |
|42.0|1.0     |
+----+--------+
```

You can check there are 4 (0,1,2,3) buckets for 5 splits

ML Feature Transformers

- StringIndexer
  - StringIndexer converts a string column to an index column
  - The most frequent label gets index 0
  - Labels are basically ordered by their frequencies

Please Note

There can be a situation when the StringIndexer may encounter a new label

This usually happens when you fit StringIndexer on one dataset and then use it to transform incoming data that may have a new label

You can use any of the following three strategies to handle the situation by setting the parameter `setHandleInvalid`to:

- 'error': throw an exception (which is the default)
- 'skip': skip the row containing the unseen label entirely
- 'keep': put unseen labels in a special additional bucket, at index numLabels

ML Feature Transformers

- Code:

```
from pyspark.ml.feature import StringIndexer
indexers = StringIndexer(inputCols= ['gender', 'heart_disease', 'smoking_history'],
                         outputCols=["gender_indexed", 'heart_disease_indexed', 'smoking_history_indexed'])
strindexedDF = indexers.fit(bucketizedDF).transform(bucketizedDF)
strindexedDF.select('gender', 'gender_indexed', 'heart_disease', 'heart_disease_indexed',
                    'smoking_history', 'smoking_history_indexed').show(5, False)
```

ML Feature Transformers

- Output:

gender	gender_indexed	heart_disease	heart_disease_indexed	smoking_history	smoking_history_indexed
Female	0.0	Yes	1.0	never	0.0
Female	0.0	No	0.0	current	2.0
Male	1.0	Yes	1.0	current	2.0
Female	0.0	No	0.0	never	0.0
Male	1.0	No	0.0	never	0.0

ML Feature Transformers

- IndexToString
  - IndexToString converts a column of label indices back to a column containing the original labels as strings
  - It is like the inverse of StringIndexer: You can retrieve the labels that were transformed by StringIndexer
  - This transformer is mostly used after training a model where you can retrieve the original labels from the prediction column

TRY IT

Class Exercise

Use IndexToString to convert index column into it respective string value

ML Feature Transformers

- OneHotEncoderEstimator
    - OneHotEncoderEstimator converts the label indices to binary vector representation with at most a single one-value
    - It represents the presence of a specific feature value from among the set of all feature values
    - It encodes the features into a sparse vector

ML Feature Transformers

- Code:

```
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder(inputCols= ["gender_indexed", 'heart_disease_indexed', 'smoking_history_indexed'],
                        outputCols=["genderVec", 'heart_diseaseVec', 'smoking_historyVec'])
encodedDF = encoder.fit(strindexedDF).transform(strindexedDF)
encodedDF.select('gender_indexed', 'genderVec', 'heart_disease_indexed', 'heart_diseaseVec',
                 'smoking_history_indexed', 'smoking_historyVec',).show(5, False)
```

ML Feature Transformers

- Output:

```
+--------------+------------+---------------------+----------------+----------------------+------------------+
|gender_indexed|genderVec   |heart_disease_indexed|heart_diseaseVec|smoking_history_indexed|smoking_historyVec|
+--------------+------------+---------------------+----------------+----------------------+------------------+
|0.0           |(2,[0],[1.0])|1.0                 |(1,[],[])       |0.0                   |(4,[0],[1.0])     |
|0.0           |(2,[0],[1.0])|0.0                 |(1,[0],[1.0])   |2.0                   |(4,[2],[1.0])     |
|1.0           |(2,[1],[1.0])|1.0                 |(1,[],[])       |2.0                   |(4,[2],[1.0])     |
|0.0           |(2,[0],[1.0])|0.0                 |(1,[0],[1.0])   |0.0                   |(4,[0],[1.0])     |
|1.0           |(2,[1],[1.0])|0.0                 |(1,[0],[1.0])   |0.0                   |(4,[0],[1.0])     |
+--------------+------------+---------------------+----------------+----------------------+------------------+
only showing top 5 rows
```

Please Note

One hot encoder in spark work very differently than the way it works in sklearn (like dummy column creation style)

Only one feature column is created representing categorical indices in the form of sparse vector in each row

You may want to convert this sparse vector to dense vector later for scaling, if required

Please Note

It is primarily used for linear model (ex. Logistic Regression) to encode categorical features since these algorithms expect continuous features

Such representations proves to be inefficient to be used with algorithms which handle categorical features intrinsically

ML Feature Transformers

- VectorAssembler
    - MLlib expects all features to be contained within a single column
    - VectorAssembler combines multiple columns and gives single column as output
    - The output column represents the values for all of the input columns in the form of vector (DenseVector or SparseVector depending on which use the least memory)

ML Feature Transformers

- Code:

```
# Import VectorAssembler from pyspark.ml.feature package
from pyspark.ml.feature import VectorAssembler
# Create a list of all the variables that you want to create feature vectors
# These features are then further used for training model
features_col = ["age", "diabetes", "hypertension", "BMI", "BodyType", "ageGroup",
                "genderVec","heart_diseaseVec","smoking_historyVec"]
# Create the VectorAssembler object
assembler = VectorAssembler(inputCols= features_col, outputCol= "features")
assembledDF = assembler.transform(encodedDF)
assembledDF.select("features").show(5)
```

ML Feature Transformers

- Output:

```
+--------------------------------------------------------+
|features                                                |
+--------------------------------------------------------+
|(13,[0,3,5,6,9],[80.0,25.19,3.0,1.0,1.0])              |
|(13,[0,3,5,6,8,11],[36.0,23.45,1.0,1.0,1.0,1.0])       |
|(13,[0,2,3,5,7,11],[76.0,1.0,20.14,3.0,1.0,1.0])       |
|(13,[0,1,3,5,6,8,9],[44.0,1.0,19.31,1.0,1.0,1.0,1.0])|
|(13,[0,3,4,5,7,8,9],[42.0,33.64,1.0,1.0,1.0,1.0,1.0])|
+--------------------------------------------------------+
only showing top 5 rows
```

If you notice, the feature column contains sparse vector

Please Note

VectorAssembler chooses dense vs sparse output format based on whichever one uses less memory

It does not convert the vector into a dense vector during the merging process

You may want to convert this feature vector, if sparse, into a dense vector to perform scaling

ML Feature Transformers

- VectorIndexer
    - VectorIndexer automatically identifies the categorical features from the feature vector (output from VectorAssembler)
    - It then indexes categorical features inside of a Vector
    - It is the vectorized version of StringIndexer
    - This step is mostly used after the VectorAssembler stage

ML Feature Transformers

- Code:

```
# Import VectorIndexer from pyspark.ml.feature package
from pyspark.ml.feature import VectorIndexer
# Create a list of all the raw features
# VectorIndexer will automatically identify the categorical columns and index them
featurecol = ['age', 'diabetes','stroke','hypertension', 'BMI','BodyType','ageGroup',
              "gender_indexed", 'heart_disease_indexed', 'smoking_history_indexed']

# Create the VectorAssembler object
assembler = VectorAssembler(inputCols= featurecol, outputCol= "features")
assembledDF = assembler.transform(strindexedDF)

# Create the VectorIndexer object. It only take feature column
vecindexer = VectorIndexer(inputCol= "features", outputCol= "indexed_features")
# Fit the vectorindexer object on the output of the vectorassembler data and transform
vecindexedDF = vecindexer.fit(assembledDF).transform(assembledDF)
vecindexedDF.select("features", "indexed_features").show(5, False)
```

ML Feature Transformers

- Output:

```
+---------------------------------------------+---------------------------------------------+
|features                                     |indexed_features                             |
+---------------------------------------------+---------------------------------------------+
|(10,[0,4,6,8],[80.0,25.19,3.0,1.0])          |(10,[0,4,6,8],[80.0,25.19,3.0,1.0])          |
|(10,[0,4,6,9],[36.0,23.45,1.0,2.0])          |(10,[0,4,6,9],[36.0,23.45,1.0,2.0])          |
|[76.0,0.0,0.0,1.0,20.14,0.0,3.0,1.0,1.0,2.0]|[76.0,0.0,0.0,1.0,20.14,0.0,3.0,1.0,1.0,2.0]|
|(10,[0,1,4,6],[44.0,1.0,19.31,1.0])          |(10,[0,1,4,6],[44.0,1.0,19.31,1.0])          |
|(10,[0,4,5,6,7],[42.0,33.64,1.0,1.0,1.0])    |(10,[0,4,5,6,7],[42.0,33.64,1.0,1.0,1.0])    |
+---------------------------------------------+---------------------------------------------+
only showing top 5 rows
```

ML Feature Transformers

Using the StringIndexer output directly as a feature will not make sense because it converts the categorical variable into nominal variable (do not have any order). Hence we one hot encode them

The VectorIndexer does the same but in the backend

Please Note

VectorIndexer let us skip the one hot encoding stage for encoding the categorical features

As discussed earlier, we should not use one hot encoding on categorical variables for algorithms like decision tree and tree ensembles

VectorIndexer are chosen over OneHotEncoderEstimator in such scenario which allows these algorithms to treat categorical features appropriately

ML Feature Transformers

- StandardScaler
  - StandardScaler scales each value in the feature vector such that the mean is 0 and the standard deviation is 1
  - It takes parameters:
    - withStd: True by default. Scales the data to unit standard deviation
    - withMean: False by default. Centers the data with mean before scaling

Please Note

To use scaling transformers, we need to assemble the features into a feature vector first (using VectorAssembler)

They do not convert sparse vector to dense vector internally. Therefore, it is very important to convert the sparse vector to a dense vector before running this step to avoid incorrect results as it does not throw error for the input sparse vector

ML Feature Transformers

- Code: We first convert sparse vector into dense vector

```
from pyspark.sql import functions as F
from pyspark.ml.linalg import Vectors, VectorUDT

# Define a udf that converts sparse vector into dense vector
# You cannot create your own custom function and run that against the data directly.
# In Spark, You have to register the function first using udf function
sparseToDense = F.udf(lambda v : Vectors.dense(v), VectorUDT())

# We then call the function here passing the column name on which the function has to be applied
densefeatureDF = assembledDF.withColumn('features_array', sparseToDense('features'))

densefeatureDF.select("features", "features_array").show(5, False)
```

ML Feature Transformers

- Output:

```
+--------------------------------------------------+------------------------------------------------------------------------+
|features                                          |features_array                                                          |
+--------------------------------------------------+------------------------------------------------------------------------+
|(13,[0,3,5,6,9],[80.0,25.19,3.0,1.0,1.0])         |[80.0, 0.0, 0.0, 25.19, 0.0, 3.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0]|
|(13,[0,3,5,6,8,11],[36.0,23.45,1.0,1.0,1.0,1.0])  |[36.0, 0.0, 0.0, 23.45, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0]|
|(13,[0,2,3,5,7,11],[76.0,1.0,20.14,3.0,1.0,1.0])  |[76.0, 0.0, 1.0, 20.14, 0.0, 3.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0]|
|(13,[0,1,3,5,6,8,9],[44.0,1.0,19.31,1.0,1.0,1.0,1.0])|[44.0, 1.0, 0.0, 19.31, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0]|
|(13,[0,3,4,5,7,8,9],[42.0,33.64,1.0,1.0,1.0,1.0,1.0])|[42.0, 0.0, 0.0, 33.64, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0]|
+--------------------------------------------------+------------------------------------------------------------------------+
only showing top 5 rows
```

ML Feature Transformers

- Code: We then apply StandardScaler on the dense vector

```
# Import StandardScaler from pyspark.ml.feature package
from pyspark.ml.feature import StandardScaler

# Create the StandardScaler object. It only take feature column (dense vector)
stdscaler = StandardScaler(inputCol= "features_array", outputCol= "scaledfeatures")

# Fit the StandardScaler object on the output of the dense vector data and transform
stdscaledDF = stdscaler.fit(densefeatureDF).transform(densefeatureDF)
stdscaledDF.select("scaledfeatures" ).show(5, False)
```

ML Feature Transformers

- Output:

```
+------------------------------------------------------------------------------------------------------------------------------------------------+
|scaledfeatures                                                                                                                                  |
+------------------------------------------------------------------------------------------------------------------------------------------------+
|[4.087032486459775,0.0,0.0,3.4969597048301693,0.0,3.465141331509375,2.0533388845793756,0.0,0.0,2.013786809426581,0.0,0.0,0.0]                  |
|[1.839164618906899,0.0,0.0,3.255407109101527,0.0,1.155047110503125,2.0533388845793756,0.0,4.588062443054963,0.0,0.0,2.888477300631736,0.0]      |
|[3.882680862136787,0.0,3.266884631584384,2.795901883893593,0.0,3.465141331509375,0.0,2.053526976254824,0.0,0.0,0.0,2.888477300631736,0.0]       |
|[2.2478678675528765,3.122394188242338,0.0,2.680678519264413,0.0,1.155047110503125,2.0533388845793756,0.0,4.588062443054963,2.013786809426581,0.0,0.0,0.0]|
|[2.145692055391382,0.0,0.0,4.670016850753747,2.080052044325287,1.155047110503125,0.0,2.053526976254824,4.588062443054963,2.013786809426581,0.0,0.0,0.0] |
+------------------------------------------------------------------------------------------------------------------------------------------------+
only showing top 5 rows
```

ML Feature Transformers

- MinMaxScaler
  - MinMaxScaler scales each value in the feature vector between 0 and 1
  - Though (0, 1) is the default range, we can define our range of max and min values as well
  - It takes parameters:
    - min: 0.0 by default. Lower bound value
    - max: 1.0 by default. Upper bound value

TRY IT

Class Exercise

Use MinMaxScaler to scale the dense features

ML Feature Transformers

- Normalizer
  - Normalizer normalize each value in the feature vector to have unit norm
  - It takes parameter p which specifies p-norm used for normalization. By default, the value of p is 2

TRY IT

Class Exercise

Use Normalizer to scale the dense features

Understanding Outputs

Understanding Output of a Model

- After you transform the dataframe with the model that you built, it may add additional columns as predictions depending upon the algorithm:
  - rawPrediction
  - probability
  - prediction

Understanding Output of a Model

- rawPrediction

- It stores the raw output of a classifier for each possible target variable label

- The meaning of a “raw” prediction may vary between algorithms

- It gives a measure of confidence in each possible label (where larger = more confident)

- For eg., for logistic regression the rawPrediction is calculated with the help of *logit*

Understanding Output of a Model

- probability

- It stores the probability of a classifier for each possible target variable label given the raw prediction

- For eg., In logistic regression, probability is the result of applying the logistic function (exp(x)/(1+exp(x))) to rawPrediction

Understanding Output of a Model

- prediction

- It is the corresponding class that the model has predicted for given *probability* array

- It takes the maximum value out of the probability array, and it gives the most probable label (single number)

Interpretation

Probability for class 0 is greater

```
+--------------------------------------+--------------------------------------+----------+
|                         rawPrediction |                          probability |prediction|
+--------------------------------------+--------------------------------------+----------+
|[0.9894187891647654,-0.9894187891647654]|[0.7289731070426124,0.2710268929573876]|       0.0 |
|[-0.9894187891647683,0.9894187891647683]|[0.271026892957387,0.728973107042613]  |       1.0 |
+--------------------------------------+--------------------------------------+----------+
```

rawPrediction for class 0

rawPrediction for class 1

probability for class 0

probability for class 1

Probability for class 1 is greater

Spark ML Algorithms

Spark ML Algorithms

- As discussed earlier, all spark ml model trains off only one column of data

- You should extract values from each row and pack them into a vector in a single column named features (name not compulsory)

- Therefore, every spark ml model has ‘featureCol’ as a parameter

- Only supervised learning models will have ‘labelCol’ along with ‘featureCol’ as a parameter

Spark ML Algorithms

Common Spark ML Parameters:

Parameter Name	Input Type	Description	Note
labelCol	Double	Target Column	Only for supervised learning algorithms
featuresCol	Vector	Features Vector	For all algorithms

Spark ML Algorithms Example: Logistic Regression

- We use LogisticRegression from pyspark.ml package to train (fit) Logistic Regression with the features

- LogisticRegression.fit returns LogisticRegressionModel object

- This object acts as a transformer that add the prediction columns to the dataframe

- This is applicable to all the spark ml algorithms

Spark ML Algorithms Example: Logistic Regression

- Code: Logistic Regression pyspark ml implementation

```
# import the LogisticRegression function from the pyspark.ml.classification package
from pyspark.ml.classification import LogisticRegression

# Build the LogisticRegression object 'lr' by setting the required parameters
lr = LogisticRegression(featuresCol="features", labelCol="label",maxIter= 10,regParam=0.3, elasticNetParam=0.8)

# fit the LogisticRegression object on the training data
lrmodel = lr.fit(trainDF)

#This LogisticRegressionModel can be used as a transformer to perform prediction on the testing data
predictonDF = lrmodel.transform(testDF)

predictonDF.select("label","rawPrediction", "probability", "prediction").show(10,False)
```

Spark ML Algorithms Example: Logistic Regression

- Output

```
+-----+----------------------------------------+----------------------------------------+----------+
|label|rawPrediction                           |probability                             |prediction|
+-----+----------------------------------------+----------------------------------------+----------+
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
|0.0  |[4.026156743176436,-4.026156743176436]|[0.9824700109051254,0.01752989094874576]|0.0       |
+-----+----------------------------------------+----------------------------------------+----------+
only showing top 10 rows
```

Interpretation

- *rawPrediction:* it is the raw output of the logistic regression classifier (array with length equal to the number of classes)

- *probability:* it is the result of applying the logistic function to rawPrediction (array of length equal to that of rawPrediction)

- *prediction:* it is the argument where the array probability takes its maximum value, and it gives the most probable label (single number)

Logistic Regression Model Evaluation

- Spark ML provides a suite of metrics for the purpose of evaluating the performance of machine learning models

- Let us evaluate the logistic regression model that we built using BinaryClassificationEvaluator

Logistic Regression Model Evaluation

- Code: Evaluating model performance using BinaryClassificationEvaluator

```
# import BinaryClassificationEvaluator from the pyspark.ml.evaluation package
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Build the BinaryClassificationEvaluator object 'evaluator'
evaluator = BinaryClassificationEvaluator()

# Calculate the accracy and print its value
accuracy = predictonDF.filter(predictonDF.label == predictonDF.prediction).count()/float(predictonDF.count())
print("Accuracy = ", accuracy)

# evaluate(predictiondataframe) gets area under the ROC curve
print('Area under the ROC curve = ', evaluator.evaluate(predictonDF))
```

```
Accuracy =  0.98134882220071
Area under the ROC curve =  0.5
```

Model Evaluation

- You can also use model.summary for logistic regression to get the performance metrics

```
# Create model summary object
lrmodelSummary = lrmodel.summary

# Print the following metrics one by one:
# 1. Accuracy
# Accuracy is a model summary parameter
print("Accuracy = ", lrmodelSummary.accuracy)
# 2. Area under the ROC curve
# Area under the ROC curve is a model summary parameter
print("Area under the ROC curve = ", lrmodelSummary.areaUnderROC)
# 3. Precision (Positive Predictive Value)
# Precision is a model summary parameter
print("Precision = ", lrmodelSummary.weightedPrecision)
# 4. Recall (True Positive Rate)
# Recall is a model summary parameter
print("Recall = ", lrmodelSummary.weightedRecall)
# 5. F1 Score (F-measure)
# F1 Score is a model summary method
print("F1 Score = ", lrmodelSummary.weightedFMeasure())
```

Model Evaluation

- Output

```
Accuracy =  0.9824700109051254
Area under the ROC curve =  0.5
Precision =  0.9652473223279173
Recall =  0.9824700109051254
F1 Score =  0.973782520813235
```

Spark ML Algorithms

Algorithm	Spark ML Package	Spark ML	Sklearn Equivalent	Output Parameter(s)
Linear Regression	pyspark.ml.regression	LinearRegression	LinearRegression	predictionCol
Logistic Regression	pyspark.ml.classification	LogisticRegression	LogisticRegression	rawPredictionCol probabilityCol predictionCol
Decision Tree Classification	pyspark.ml.classification	DecisionTreeClassifier	DecisionTreeClassifier	rawPredictionCol probabilityCol predictionCol
Decision Tree Regression	pyspark.ml.regression	DecisionTreeRegressor	DecisionTreeRegressor	predictionCol

Spark ML Algorithms

Algorithm	Spark ML Package	Spark ML	Sklearn Equivalent	Output Parameter(s)
Random Forest Classification	pyspark.ml.classification	RandomForestClassifier	RandomForestClassifier	rawPredictionCol probabilityCol predictionCol
Random Forest Regression	pyspark.ml.regression	RandomForestRegressor	RandomForestRegressor	predictionCol
Gradient Boosted Trees Classification	pyspark.ml.classification	GBTClassifier	GradientBoostingClassifier	rawPredictionCol probabilityCol predictionCol
Gradient Boosted Trees Regression	pyspark.ml.regression	GBTRegressor	GradientBoostingRegressor	predictionCol

Spark ML Algorithms

Algorithm	Spark ML Package	Spark ML	Sklearn Equivalent	Output Parameter(s)
Support Vector Machines (SVM)	pyspark.ml.classification	LinearSVC	LinearSVC	rawPredictionCol probabilityCol predictionCol
Naive Bayes	pyspark.ml.classification	NaiveBayes	GaussianNB	rawPredictionCol probabilityCol predictionCol
K-means	pyspark.ml.clustering	KMeans	GradientBoostingClassifier	predictionCol

Model Evaluation

- Following evaluators are available in pyspark.ml.evaluation package

Evaluator	Metric Available
BinaryClassificationEvaluator	areaUnderROC areaUnderPR
MulticlassClassificationEvaluator	f1, accuracy, weightedPrecision, weightedRecall, weightedTruePositiveRate, weightedFalsePositiveRate, weightedFMeasure, truePositiveRateByLabel, falsePositiveRateByLabel, precisionByLabel, recallByLabel, fMeasureByLabel, logLoss, hammingLoss

Model Evaluation

Evaluator	Metric Available
RegressionEvaluator	rmse, mse, r2, mae, var
MultilabelClassificationEvaluator	subsetAccuracy, accuracy, hammingLoss, precision, recall, f1Measure, precisionByLabel, recallByLabel, f1MeasureByLabel, microPrecision, microRecall, microF1Measure
ClusteringEvaluator	silhouette

Building Pipeline

Building Spark ML Pipeline

- As discussed earlier, a spark pipeline is a sequence of Transformers and an Estimator

- These stages run in order and the dataframe is transformed as it passes through each stage

- We will now see how to build a pipeline in pyspark

Building Spark ML Pipeline

- To build a pipeline we import the Pipeline module from pyspark.ml package
- Next, we create a pipeline object by passing all transformers and an estimator as a list of stages
- This object is later fit on the raw training set, which creates a pipeline model
- This model is later used as a transformer to be applied on testing set to make predictions

Building Spark ML Pipeline

- Code: Building and implementing a spark ml pipeline

```
# import Pipeline from pyspark.ml package
from pyspark.ml import Pipeline

# Build the pipeline object by providing stages(transformers + Estimator)
# that you need the dataframe to pass through
# Transfoermers - binarizer, bucketizer, indexers, encoder, assembler
# Estimator - lr
lrpipeline = Pipeline(stages=[binarizer, bucketizer, indexers, encoder, assembler, lr])

# fit the pipeline for the trainind data
lrpipelinemodel = lrpipeline.fit(trainDF)

# transform the data
lrpipelinepredicted = lrpipelinemodel.transform(testDF)

# view some of the columns generated
lrpipelinepredicted.select('label', 'rawPrediction', 'probability', 'prediction').show()
```

Building Spark ML Pipeline

- Output

```
+-----+--------------------+--------------------+----------+
|label|       rawPrediction|         probability|prediction|
+-----+--------------------+--------------------+----------+
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
+-----+--------------------+--------------------+----------+
only showing top 10 rows
```

Model Persistence

Model Persistence

- In real-life scenarios, you will be producing ML model and hands it over to the development team for deploying in a production environment

- This becomes easier with model persistence

- Model persistence means saving your model to a disk for later use without the need to retrain your model

Model Persistence

- We use model.save('path') to save our model at the desired location
- It might happen that you wish to retrain your model and save it at the same the place
- In those cases, use model.write().overwrite().save('path') to save your retrained model at the same place

Model Persistence

- Code - Saving the model

```
# use save() method to save the model
# write().overwrite() is usually used when you want to replace the older model with a new one
# It might happen that you wish to retrain your model and save it at the same the place
lrpipelinemodel.write().overwrite().save("/FileStore/models/lrmodel")
```

Model Persistence

- You can then load the model and perform predictions

- Use PipelineModel module from pyspark.ml package to load the persisted pipeline model

- The loaded model can then be used for perform prediction on test data

Model Persistence

- Code: Loading the model

```
# import PipelineModel from pyspark.ml package
from pyspark.ml import PipelineModel

# load the model from the location it is stored
# The loaded model acts as PipelineModel
pipemodel = PipelineModel.load("/FileStore/models/lrmodel")

# use the PipelineModel object to perform prediciton on test data.
# Use .transform() to perfrom prediction
prediction = pipemodel.transform(testDF)

# print the results
prediction.select('label', 'rawPrediction', 'probability', 'prediction').show(5)
```

Model Persistence

- Output

```
+-----+--------------------+--------------------+----------+
|label|       rawPrediction|         probability|prediction|
+-----+--------------------+--------------------+----------+
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
|  0.0|[4.02615674317643...|[0.98247001090512...|       0.0|
+-----+--------------------+--------------------+----------+
only showing top 5 rows
```

Summary

- Spark MLlib is Apache Spark's *M*achine *L*earning *lib*rary
- *spark.mllib* package built on top of the RDD API
- *spark.ml* package built on top of higher-level DataFrame API
- Pipeline API chains Transformers and Estimator each as a stage to specifying ML workflow
- Spark ML library provides number of transformers to preprocess the data

Thank You