Big data: architectures and data analytics

Spark MLlib

Spark MLlib

- Spark MLlib is the Spark component providing the machine learning/data mining algorithms
  - Pre-processing techniques
  - Classification (supervised learning)
  - Clustering (unsupervised learning)
  - Model selection and tuning
  - ...

Guide available at:
https://spark.apache.org/docs/latest/ml-guide.html

Spark MLlib

- MLlib APIs are divided into two packages:
  - **pyspark.mllib**
    - It contains the original APIs built on top of RDDs
    - This version of the APIs is in maintenance mode and will be probably deprecated in the next releases of Spark
  - **pyspark.ml**
    - It provides higher-level API built on top of DataFrames for constructing ML pipelines
    - It is recommended because the DataFrame-based API is more versatile and flexible
    - It provides the pipeline concept

Spark MLlib

- MLlib APIs are divided into two packages:
  - **pyspark.mllib**
    - It contains the original APIs built on top of RDDs
    - This version of the APIs is in maintenance mode and will be probably deprecated in the next releases of Spark
  - **pyspark.ml**
    - It provides higher-level API built on top of DataFrames for constructing ML pipelines
    - It is recommended because the DataFrame-based API is more versatile and flexible
    - It provides the pipeline concept

Spark MLlib – Data types

Spark MLlib – Data types

- Spark MLlib is based on a set of basic local and distributed data types
  - Local vector
  - Labeled point
  - Local matrix
  - Distributed matrix
  - ..
- DataFrames for ML contain objects based on those basic data types
- Often they will be transparent to you, but it’s good to understand what’s happening under the hood!

Local vectors

- Local **pyspark.mllib.linalg.Vector** objects in MLlib are used to store vectors of double values
  - Dense and sparse vectors are supported
- The MLlib algorithms work on vectors of doubles
  - Vectors of doubles are used to represent the input records/data
    - One vector for each input record
  - Non double attributes/values must be mapped to double values

Local vectors

- Dense and sparse representations are supported
- E.g., a vector (1.0, 0.0, 3.0) can be represented
  - in dense format as [1.0, 0.0, 3.0]
  - or in sparse format as (3, [0, 2], [1.0, 3.0])
    - where 3 is the size of the vector
    - The array [0,2] contains the indexes of the non-zero cells
    - The array [1.0, 3.0] contains the values of the non-zero cells

Local vectors

- The following code shows how a vector can be created in Spark

```
from pyspark.mllib.linalg import Vectors


#Create a dense vector (1.0, 0.0, 3.0).
dv = Vectors.dense(1.0, 0.0, 3.0)

#Create a sparse vector (1.0, 0.0, 3.0) by
#specifying its indices and values corresponding
#to non-zero entries
sv = Vectors.sparse(3, [0, 2], [1.0, 3.0])
```

Local vectors

- The following code shows how a vector can be created in Spark

```
from pyspark.mllib.linalg import Vectors


#Create a dense vector (1.0, 0.0, 3.0)
dv = Vectors.dense(1.0, 0.0, 3.0)

#Create a sparse vector (1.0, 0.0, 3.0) by
#specifying its indices and values corresponding
#to non-zero entries
sv = Vectors.sparse(3, [0, 2], [1.0, 3.0])
```

Size of the vector

Indexes of non-empty cells

Values of non-empty cells

Labeled points

Local **pyspark.mllib.regression.LabeledPoint** objects are local vectors of doubles associated with a label

- The label is a double value
  - For the classification problem, each class label is associated with an integer value (casted to a double) ranging from 0 to C-1, where C is the number of distinct classes
  - For the regression problem, the label is the real value to predict
- Both dense and sparse vectors associated with a label are supported

Labeled points

LabeledPoint objects are created by invoking the **LabeledPoint(double label, SparseVector features)** constructor

- Note that label is a double and also Vector is a vector of doubles
- features can be also NumPy array or a list

Labeled points

- In MLlib, labeled points are used by supervised (classification and regression) machine learning algorithms to represent records/data points
  - The **label** part represents the **target** of the analysis
  - The **features** part represents the **predictive attributes** that are used to predict the target attribute, i.e., the value of label

Labeled points: classification example

- Suppose the analyzed records/data points are characterized by
  - 3 real (predictive) attributes/features
  - A class label attribute that can assume two values: 0 or 1
    - This is a binomial classification problem
- We want to predict the value of the class label attribute based on the values of the other attributes/features

Labeled points: classification example

- Consider the following two records/data points
  - Attributes/features = [2.0,5.0,3.0] -- Label = 0
  - Attributes/features = [1.0,0.0,3.0] -- Label = 1
- Two LabeledPoint objects to represent those two data points in Spark

Labeled points: classification example

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense
    feature vector.
neg = LabeledPoint(0.0, [2.0, 5.0, 3.0])

# Create a labeled point with a negative label and a sparse
    feature vector.
pos = LabeledPoint(1.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Labeled points: classification example

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense
    feature vector.
neg = LabeledPoint(0.0, [2.0, 5.0, 3.0])

# Create a labeled point with a negative label and a sparse
    feature vector.
pos = LabeledPoint(1.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

Value of the class label

Labeled points: classification example

```
from pyspark.mllib.linalg
from pyspark.mllib.regressi

# Create a labeled point with a positive label and a dense
    feature vector.
neg = LabeledPoint(0.0, [2.0, 5.0, 3.0])

# Create a labeled point with a negative label and a sparse
    feature vector.
pos = LabeledPoint(1.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

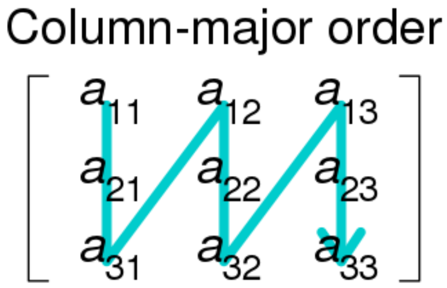
Vector of doubles representing the values of the predictive features/attributes

Local Matrix

Local **pyspark.ml.linalg.Matrices** objects are matrices

- Integer-typed row and column indices
- Double-typed values

- Both dense and sparse matrices are supported
- Explicit the column-major order



Local Matrix

- Consider the following matrix

$$\begin{bmatrix} 9 & 0 \\ 0 & 8 \\ 0 & 6 \end{bmatrix}$$

- Build a local matrix, both in dense and sparse format

Local Matrix

```
from pyspark.mllib.linalg import Matrices

# Create a dense matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
dm = Matrices.dense(3, 2, [9, 0, 0, 0, 8, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0,
   6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Local Matrix

Number of rows and columns

```
from pyspark.mllib.linalg import Matrices

# Create a dense matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
dm = Matrices.dense(3, 2, [9, 0, 0, 0, 8, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0,
   6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Local Matrix

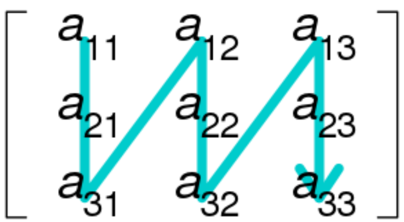
from pyspark.mllib.linalg in

Values to fill: first column, second column,...

```
# Create a dense matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
dm = Matrices.dense(3, 2, [9, 0, 0, 0, 8, 6])

# Create a sparse matrix ((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 2, 1], [9, 6, 8])
```

Values to fill: indexes of new column, row indexes of values, values

Column-major order



Sparse labeled data

Sparse labeled data: The LIBSVM format

- Frequently the training data are sparse
  - E.g., textual data are sparse
    - Each document contains only a subset of the possible words
  - Hence, sparse vectors are frequently used
- MLlib supports reading training examples stored in the LIBSVM format
  - It is a commonly used textual format that is used to represent sparse documents/data points

Sparse labeled data: The LIBSVM format

- The LIBSVM format
  - It is a textual format in which each line represents a labeled point by using a sparse feature vector:
- Each line has the format
  label index1:value1 index2:value2 ...
- where
  - label is an integer associated with the class label
    - It is the first value of each line
  - The indexes are integer values representing the features
  - The values are the (double) values of the features

Sparse labeled data: The LIBSVM format

- Consider the following two records/data points characterized by 4 predictive features and a class label
  - Features = [5.8, 1.7, 0 , 0] -- Label = 1
  - Features = [4.1, 0 , 2.5, 1.2] -- Label = 0
- Their LIBSVM format-based representation is the following

```
1 1:5.8 2:1.7
0 1:4.1 3:2.5 4:1.2
```

Sparse labeled data: The LIBSVM format

- LIBSVM files can be loaded into DataFrames by combining the following methods:
  - read(), format("libsvm"), and load(String inputpath)
- The returned DataFrame has two columns:
  - label: double
    - The double value associated with the label
  - features: vector
    - A sparse vector associated with the predictive features

Sparse labeled data: The LIBSVM format

Input sample_libsvm_data.txt

```
1 1:5.8 2:1.7
0 1:4.1 3:2.5 4:1.2
```

```
myDF =
   spark.read.format("libsvm").load("sample_libsvm_data.txt")
```

Output

```
+-----+------------------------+
|label|features                |
+-----+------------------------+
|1.0  |(4,[0,1],[5.8,1.7])     |
|0.0  |(4,[0,2,3],[4.1,2.5,1.2])|
+-----+------------------------+
```

Sparse labeled data: The LIBSVM format

Input sample_libsvm_data.txt

```
1 1:5.8 2:1.7
0 1:4.1 3:2.5 4:1.2
```

```
myDF =
   spark.read.format("libsvm").load("sample_libsvm_data.txt")
```

Name of the columns automatically assigned

Output

```
+-----+------------------------+
|label|features                |
+-----+------------------------+
|1.0  |(4,[0,1],[5.8,1.7])     |
|0.0  |(4,[0,2,3],[4.1,2.5,1.2])|
+-----+------------------------+
```

Sparse labeled data: The LIBSVM format

Input sample_libsvm_data.txt

```
1 1:5.8 2:1.7
0 1:4.1 3:2.5 4:1.2
```

myDF = spark.read.format("libsvm").load("sample_libsvm_data.txt")

Output

Sparse Vectors

```
+-----+-----------------------+
|label|features               |
+-----+-----------------------+
|1.0  |(4,[0,1],[5.8,1.7])    |
|0.0  |(4,[0,2,3],[4.1,2.5,1.2])|
+-----+-----------------------+
```

Spark MLlib - Main concepts

Spark MLlib - Main concepts

- Spark MLlib (**pyspark.ml**) uses DataFrames as input data
- Hence, the input of the MLlib algorithms are structured data (i.e., tables)
- All input data must be represented by means of “tables” before applying the MLlib algorithms
  - Also the document collections must be transformed in a tabular format

Spark MLlib - Main concepts

- The DataFrames used by the MLlib algorithms are characterized by several columns associated with different characteristics of the input data. Among them:
  - label
    - Target of a classification/regression analysis
  - features
    - A vector containing the values of the attributes/features of the input record/data points
  - ..

Spark MLlib - Main concepts

- Transformer
  - A Transformer is an ML algorithm/procedure that transforms a DataFrame into another DataFrame
    - It has the method **transform()**
    - E.g., a classification model is a Transformer that can be applied on a DataFrame with features and transforms it into a DataFrame with also the prediction column
    - E.g., 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

Spark MLlib - Main concepts

- Estimator
  - An Estimator is a ML algorithm/procedure that is applied on a DataFrame to produce a Transformer (i.e., a model)
    - Each Estimator implements a method **fit()**
    - It accepts a DataFrame and produces a **Model of type Transformer**
  - An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on an input dataset and returns a model
    - E.g., the Decision Tree classification algorithm is an Estimator, and calling fit() on it a Decition Tree is built, which is a Model and hence a Transformer

Spark MLlib - Main concepts

- Pipeline
  - A Pipeline chains multiple Transformers and Estimators together to specify a Machine learning/Data Mining workflow
    - The output of a transformer/estimator is the input of the next one in the pipeline
  - E.g., a simple text document processing workflow aiming at building a classification model includes several steps
    - Split each document into a set of words
    - Convert each set of words into a numerical feature vector
    - Learn a prediction model using the feature vectors and the associated class labels

Spark MLlib - Main concepts

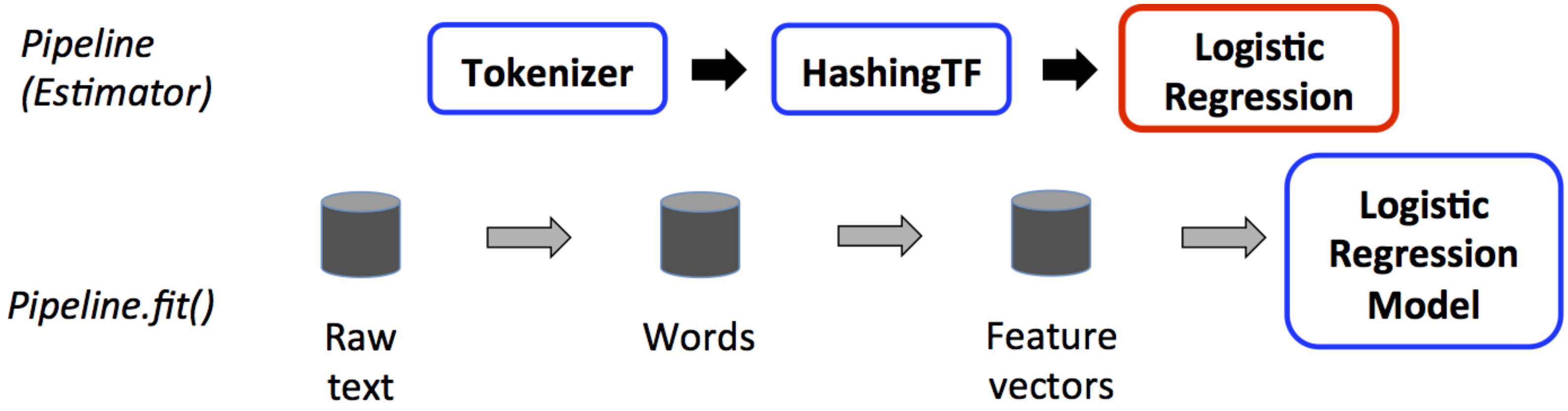
- Parameters
  - All Transformers and Estimators share common APIs for specifying parameters

Spark MLlib - Main concepts

- In the new APIs of Spark MLlib the use of the pipeline approach is preferred/recommended
- This approach is based on the following steps
  1. The set of Transformers and Estimators that are needed are instantiated
  2. A pipeline object is created and the sequence of transformers and estimators associated with the pipeline are specified
  3. The pipeline is executed and a model is created
  4. (optional) The model is applied on new data

Spark MLlib - Main concepts

- Pipeline for the estimator to create the model



Spark MLlib – Main concepts

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Spark MLlib – Main concepts

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Pipeline concatenates different estimators

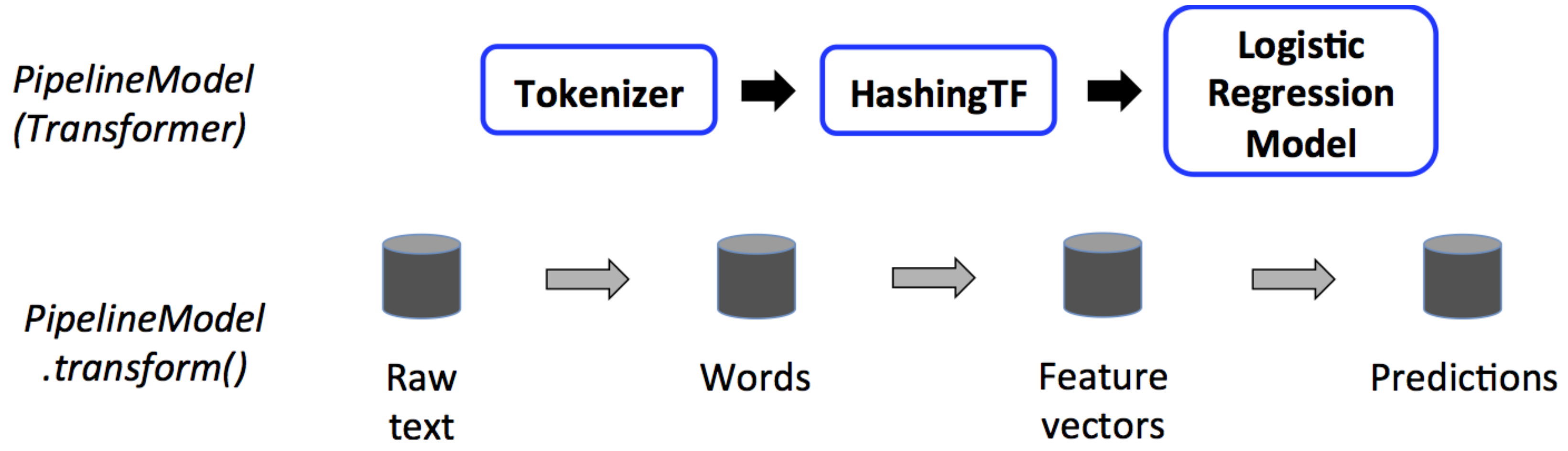
Spark MLlib – Main concepts

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)
```

Estimators will be fitted (all in the pipeline) and return a transformers

Spark MLlib - Main concepts

- Pipeline for the transformer to make the predictions



Spark MLlib – Main concepts

```
# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
```

Spark MLlib – Main concepts

```
# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
```

The built transformer is used to transform a DF into another DF

Splitting the data

Split into train and test

- We have our data saved in a DataFrame
- We want to split the dataframe in order to have test data that we will use only at the end to evaluate performance of the ML classification/regression/…
- The rest of the data will be used for preprocessing (fitting), training and validation
- Use the method **randomSplit(weights,seed)** of DataFrame

Split into train and test

- **Remember**: you must NOT fit anything on the test set!
- The test can be only transformed!

Split into train and test

- Method **randomSplit(weights,seed)** of DataFrame
- Parameters:
  - weights – list of doubles as weights with which to split the DataFrame. Weights will be normalized if they don’t sum up to 1.0
  - seed – The seed for sampling
- Output: as many DataFrames as len(weights)

Split into train and validation

Input DataFrame

Bytes Transferred	Pages_Corrupted	Location
391.09	7.0	Slovenia
720.99	9.0	British Virgin Is...
356.32	8.0	Tokelau
228.08	8.0	Bolivia
408.5	8.0	Iraq
390.69	9.0	Marshall Islands
342.97	7.0	Georgia
101.61	7.0	Timor-Leste
275.53	8.0	Palestinian Terri...
424.83	8.0	Bangladesh
249.09	9.0	Northern Mariana ...
242.48	8.0	Zimbabwe
514.54	8.0	Isle of Man
284.77	9.0	Sao Tome and Prin...
779.25	8.0	Greece
307.31	7.0	Solomon Islands
355.94	7.0	Guinea-Bissau
372.65	7.0	Burkina Faso
347.23	7.0	Mongolia
...	...	...

334 rows

Split into train and test

```
#splitting dataframe

inputDF=spark.read.csv('hack_data.csv',header=True,inf
    erSchema=True)

print(inputDF.count())

trainValidation, test= inputDF.randomSplit([0.75, 0.25])

print(trainValidation.count())
print(test.count())
print(trainValidation.count()+test.count())
```

Split into train and test

We take around 75% of the data for train/validation and 25% for test

```
#splitting dataframe

inputDF=spark.read.csv('hack_data.csv',header=True,inferSchema=True)

print(inputDF.count())

trainValidation, test= inputDF.randomSplit([0.75, 0.25])

print(trainValidation.count())
print(test.count())
print(trainValidation.count()+test.count())
```

Split into train and test

```
#splitting dataframe

inputDF=spark.read.csv('hac
    erSchema=True)

print(inputDF.count())

trainValidation, test= inputDF.randomSplit([0.75, 0.25])

print(trainValidation.count())
print(test.count())
print(trainValidation.count()+test.count())
```

Note: without fixing the seed the output dataframes content will change, as well as their number of lines

Split into train and test

Input DataFrame

Bytes Transferred	Pages_Corrupted	Location
391.09	7.0	Slovenia
720.99	9.0	British Virgin Is...
356.32	8.0	Tokelau
228.08	8.0	Bolivia
408.5	8.0	Iraq
390.69	9.0	Marshall Islands
342.97	7.0	Georgia
101.61	7.0	Timor-Leste
275.53	8.0	Palestinian Terri...
424.83	8.0	Bangladesh
249.09	9.0	Northern Mariana ...
242.48	8.0	Zimbabwe
514.54	8.0	Isle of Man
284.77	9.0	Sao Tome and Prin...
779.25	8.0	Greece
307.31	7.0	Solomon Islands
355.94	7.0	Guinea-Bissau
372.65	7.0	Burkina Faso
347.23	7.0	Mongolia

334 rows

“trainValidation” DataFrame

Bytes Transferred	Pages_Corrupted	Location
10.0	8.0	Sri Lanka
23.3	7.0	Sao Tome and Prin...
39.13	9.0	Romania
52.79	7.0	Equatorial Guinea
60.16	8.0	Sri Lanka
64.83	8.0	Saudi Arabia
67.17	7.0	Faroe Islands
75.53	8.0	Wallis and Futuna
89.49	8.0	Holy See (Vatican...
91.86	8.0	Yemen
101.61	7.0	Timor-Leste
104.67	9.0	Cyprus
106.26	8.0	Netherlands Antilles
110.81	8.0	Jamaica
118.03	7.0	Australia
125.69	7.0	Ukraine
169.89	9.0	Armenia
178.56	8.0	Sweden
179.55	8.0	Azerbaijan

255 rows

“test” DataFrame

Bytes Transferred	Pages_Corrupted	Location
11.04	8.0	Botswana
84.83	9.0	South Africa
171.73	8.0	Bouvet Island (Bo...
228.08	8.0	Bolivia
256.61	7.0	Belize
272.34	9.0	Norway
281.56	9.0	Saint Kitts and N...
309.84	9.0	Zambia
316.42	9.0	South Africa
320.47	8.0	Montenegro
334.8	8.0	Netherlands
345.41	9.0	Portugal
347.23	7.0	Mongolia
348.41	9.0	Uruguay
354.77	8.0	Netherlands Antilles
356.32	8.0	Tokelau
364.88	7.0	Philippines
366.49	9.0	China
369.0	7.0	Israel

79 rows

How to improve results?

- When we train an algorithm we have to take care that the model is not over-specified to the specific input data
- We should validate our results on other data
- Usually the data are again split into **training** and **validation**
- We validate results and tune parameters with the validation set
- More on this topic later in “parameter tuning” section

Data pre-processing

Feature transformation

Many useful methods of **pyspark.ml.feature**

- They will be inserted into a Pipeline for performing ML
- Some are mandatory to preprocess the data to make the ML work
- We will see some of them!

Others available at:

http://spark.apache.org/docs/latest/ml-features.html

Vector Assembler

VectorAssembler (**pyspark.ml.feature.VectorAssembler**) is a transformer that combines a given list of columns into a single vector column

- Useful for combining features into a single feature vector, in order to train ML models
- 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.

Vector Assembler

Example.csv

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True

ColA has integers
ColB has double
ColC has Booleans

Vector Assembler

```
#Importing VectorAssembler and creating our Features
# “Features” is  single column where each row of the DataFrame
    contains a feature vector.

from pyspark.ml.feature import VectorAssembler

inputDF=spark.read.csv('example.csv',header=True,inferSchema=Tr
    ue)

feat_cols = ['colA', 'colB', 'colC']
vectorAssembler = VectorAssembler(inputCols = feat_cols,
    outputCol = 'features')

transformedDF = vectorAssembler.transform(inputDF)
```

Vector Assembler

```
#Importing VectorAssembler and creating our Features
# "Features" is  single column where each row of the DataFrame
    contains a feature vector.

from pyspark.ml.feature import Vec

inputDF=spark.read.csv('example.c
    ue)

feat_cols = ['colA', 'colB', 'colC']
vectorAssembler = VectorAssembler(inputCols = feat_cols,
    outputCol = 'features')

transformedDF = vectorAssembler.transform(inputDF)
```

The columns to "assemble" and the name of the output column are defined

Vector Assembler

```
#Importing VectorAssembler and creating our Features
# “Features” is  single column where each row of the DataFrame contains a feature vector.

from pyspark.ml.feature import VectorAssembler

inputDF=spark.read.csv('example.csv',header=True,inferSchema=True)

feat_cols = ['colA', 'colB', 'colC']
vectorAssembler = VectorAssembler(inputCols = feat_cols, outputCol = 'features')

transformedDF = vectorAssembler.transform(inputDF)
```

The built transformer is used to transform a DF into another DF

Vector Assembler

Example.csv

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True

A column of DataFrame can be also a Vector

Transformed dataframe

colA	colB	colC	features
1	4.5	True	[1.0,4.5,1.0]
2	0.6	True	[2.0,0.6,1.0]
3	1.5	False	[3.0,1.5,0.0]
4	12.1	True	[4.0,12.1,1.0]
5	0.0	True	[5.0,0.0,1.0]

Standard scaler

StandardScaler (**pyspark.ml.feature.StandardScaler**) transforms a dataset of **Vector rows**, normalizing each feature to have unit standard deviation and/or zero mean. Important pre-processing for improving the performance of many ML algorithms

- is an **Estimator** which can be **fit** on a dataset to produce a StandardScalerModel;
- The model can then transform a Vector column of doubles
- withStd: True by default. Scales the data to unit standard deviation.
- withMean: False by default. Centers the data with mean before scaling. It will build a dense output, so take care when applying to sparse input.

Standard scaler

Example.csv with created column “features”

```
+----+----+-----+--------------+
|colA|colB| colC|      features|
+----+----+-----+--------------+
|   1| 4.5| true| [1.0,4.5,1.0]|
|   2| 0.6| true| [2.0,0.6,1.0]|
|   3| 1.5|false| [3.0,1.5,0.0]|
|   4|12.1| true|[4.0,12.1,1.0]|
|   5| 0.0| true| [5.0,0.0,1.0]|
+----+----+-----+--------------+
```

Standard scaler

#Importing the StandardScaler Library and Creating Scaler
#Centering and Scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the transform method.
#Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature does not more or less look like standard normally distributed data.

```
from pyspark.ml.feature import StandardScaler

scaler = StandardScaler(inputCol='features', outputCol="scaledFeatures",
    withStd=True, withMean=True)

# fitting the StandardScaler. Then Normalize each feature to have a unit
    standard deviation.
scalerModel = scaler.fit(transformedDF)

scaledDF = scalerModel.transform(transformedDF)
```

Standard scaler

```
#Importing the StandardScaler Library and Creating Scaler
#Centering and Scaling happen independently on each feature by computing
    the relevant statistics on the samples in the training set. Mean and standard
    deviation are then stored to be used on later data using the transform
    method.
#Standardization of a dataset is a common requirement for many machine
    learning estimators: they might behave badly if the individual feature does
    not more or less look like standard nor

from pyspark.ml.feature import Standard

scaler = StandardScaler(inputCol='features', outputCol="scaledFeatures",
    withStd=True, withMean=True)

# fitting the StandardScaler. Then Normalize each feature to have a unit
    standard deviation.
scalerModel = scaler.fit(transformedDF)

scaledDF = scalerModel.transform(transformedDF)
```

The column to “standardize” is defined

Standard scaler

#Importing the StandardScaler Library and Creating Scaler
#Centering and Scaling happen independently on each feature by computing the relevant statistics on the samples in the training set. Mean and standard deviation are then stored to be used on later data using the transform method.
#Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual feature does not more or less look like standard nor

```
from pyspark.ml.feature import Standard

scaler = StandardScaler(inputCol='features', outputCol="scaledFeatures",
    withStd=True, withMean=True)

# fitting the StandardScaler. Then Normalize each feature to have a unit
    standard deviation.
scalerModel = scaler.fit(transformedDF)

scaledDF = scalerModel.transform(transformedDF)
```

The transformer “scalerModel” is created and then it is used to create the new DataFrame “scaledDF”

Standard scaler

Example.csv with created column "features"

```
+----+----+-----+--------------+
|colA|colB| colC|      features|
+----+----+-----+--------------+
|   1| 4.5| true| [1.0,4.5,1.0]|
|   2| 0.6| true| [2.0,0.6,1.0]|
|   3| 1.5|false| [3.0,1.5,0.0]|
|   4|12.1| true|[4.0,12.1,1.0]|
|   5| 0.0| true| [5.0,0.0,1.0]|
+----+----+-----+--------------+
```

Scaled (transformed) dataframe

```
+----+----+-----+--------------+-------------------------------------------------------------+
|colA|colB|colC |features      |scaledFeatures                                               |
+----+----+-----+--------------+-------------------------------------------------------------+
|1   |4.5 |true |[1.0,4.5,1.0] |[-1.2649110640673518,0.1525102377187456,0.44721359549995787] |
|2   |0.6 |true |[2.0,0.6,1.0] |[-0.6324555320336759,-0.6301080874169228,0.44721359549995787]|
|3   |1.5 |false|[3.0,1.5,0.0] |[0.0,-0.4495038585394609,-1.788854381999832]                 |
|4   |12.1|true |[4.0,12.1,1.0]|[0.6324555320336759,1.677612614906202,0.44721359549995787]   |
|5   |0.0 |true |[5.0,0.0,1.0] |[1.2649110640673518,-0.7505109066685641,0.44721359549995787] |
+----+----+-----+--------------+-------------------------------------------------------------+
```

Tokenizer

Tokenizer (**pyspark.ml.feature.Tokenizer**) is the process of taking text (such as a sentence) and breaking it into individual terms (usually words).

- is a Transformer that take in input a Vector column of strings and return a Vector column of Vectors of strings ;

Tokenizer

Example DataFrame

```
+---+------------------------------------+
|id |sentence                            |
+---+------------------------------------+
|0  |Hi I heard about Spark              |
|1  |I wish we can have more Spark classes|
+---+------------------------------------+
```

Tokenizer

```
#Importing the Tokenizer Library and creating a Tokenizer

from pyspark.ml.feature import Tokenizer

sentenceDF = spark.createDataFrame([
    (0, "Hi I heard about Spark"),
    (1, "I wish we can have more Spark classes"),
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence",
    outputCol="words")
tokenizedDF = tokenizer.transform(sentenceDF)
```


74

Tokenizer

```
#Importing the Tokenizer Library and creating a Tokenizer

from pyspark.ml.feature import Tokenizer

sentenceDF = spark.createData
    (0, "Hi I heard about Spark"),
    (1, "I wish we can have more
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence",
    outputCol="words")
tokenizedDF = tokenizer.transform(sentenceDF)
```

Define the column to transform “sentence” and the name of the new column “words”

75

Tokenizer

Example DataFrame

```
+---+-----------------------------------+
|id |sentence                           |
+---+-----------------------------------+
|0  |Hi I heard about Spark             |
|1  |I wish we can have more Spark classes|
+---+-----------------------------------+
```

Tokenized (tranformed) DataFrame

```
+---+-------------------------------------+----------------------------------------------+
|id |sentence                             |words                                         |
+---+-------------------------------------+----------------------------------------------+
|0  |Hi I heard about Spark               |[hi, i, heard, about, spark]                  |
|1  |I wish we can have more Spark classes|[i, wish, we, can, have, more, spark, classes]|
+---+-------------------------------------+----------------------------------------------+
```

StringIndexer / IndexToString

- Frequently the **class label or a feature** is a **categorical value** (i.e., a string)
- Spark MLlib works only with numerical values and hence categorical class label of feature values **must be mapped to integer (and then double) values**

StringIndexer / IndexToString

- The Estimators **StringIndexer** and **IndexToString** support the transformation of categorical class label into numerical one
  - StringIndexer maps each categorical value of the class label to an integer (finally casted to a double)
  - IndexToString is used to perform the opposite operation

StringIndexer / IndexToString

Input DataFrame

categoricalLabel	features
Positive	[0.0, 1.1, 0.1]
Negative	[2.0, 1.0, -1.0]
Negative	[2.0, 1.3, 1.0]

Transformed DataFrame

categorical Label	label	features
Positive	1.0	[0.0, 1.1, 0.1]
Negative	0.0	[2.0, 1.0, -1.0]
Negative	0.0	[2.0, 1.3, 1.0]

StringIndexer / IndexToString

Input Da

The categorical values of categoricalLabel (the class label column) must be mapped to integer values (finally casted to doubles)

categoricalLabel	features
Positive	[0.0, 1.1, 0.1]
Negative	[2.0, 1.0, -1.0]
Negative	[2.0, 1.3, 1.0]

Transformed DataFrame

categorical Label	label	features
Positive	1.0	[0.0, 1.1, 0.1]
Negative	0.0	[2.0, 1.0, -1.0]
Negative	0.0	[2.0, 1.3, 1.0]

StringIndexer / IndexToString

```
from pyspark.ml.feature import StringIndexer

inputDF=spark.read.csv('StringIndexer.txt',header=True,inferSchema=True)

indexer = StringIndexer(inputCol="categoricalLabel", outputCol="label")
indexerModel = indexer.fit(inputDF)
indexedDF=indexerModel.transform(inputDF)
```

StringIndexer / IndexToString

```
from pyspark.ml.feature import StringIndexer

inputDF=spark.read.csv('StringIndexer.txt',header=True,infer
    Schema=True)

indexer = StringIndexer(inputCol="categoricalLabel",
    outputCol="label")
indexerModel = indexer.fit(inputDF)
indexedDF=indexerModel.transform(inputDF)
```

Define the categorical column to transform “categoricalLabel” and the name of the new column “label”

String Indexer / IndexToString

- Symmetrically to StringIndexer, IndexToString maps a column of label indices back to a column containing the original labels as strings.
- A common use case is to produce indices from labels with StringIndexer, train a model with those indices and retrieve the original labels from the column of predicted indices

StringIndexer / IndexToString

Input new DataFrame

label	features
1.0	[45.0, 0.1, 2.1]
1.0	[22.1, 2.0, -1.2]
0.0	[3.0, 11.4, 1.9]

Transformed DataFrame

label	features	originalLabel
1.0	[45.0, 0.1, 2.1]	Positive
1.0	[22.1, 2.0, -1.2]	Positive
0.0	[3.0, 11.4, 1.9]	Negative

StringIndexer / IndexToString

```
from pyspark.ml.feature import IndexToString

converter = IndexToString(inputCol="label",
    outputCol="originalLabel")

convertedDF = converter.transform(indexedDF)
```

StringIndexer / IndexToString

```
from pyspark.ml.feature import IndexToString

converter = IndexToString(inputCol="label",
    outputCol="originalLabel")

convertedDF = converter.transform(indexedDF)
```

InputCol should be the outputCol of a previous StringIndexer

OneHotEncoderEstimator

- One-hot encoding maps a **categorical feature**, represented as an index, **to a binary vector** with at most **a single one-value** indicating the presence of a specific feature value
- This encoding allows algorithms which expect continuous features, such as Decision Trees, to correctly use categorical features.
- For string type input data, it is common to encode categorical features using StringIndexer first.
- Output vectors are saved as **sparse vectors** (by definition they have at maximum a single one-value)

OneHotEncoderEstimator

- OneHotEncoderEstimator can **transform multiple columns**, returning an one-hot-encoded output vector column for each input column. It is common to merge these vectors into a single feature vector using VectorAssembler.
- OneHotEncoderEstimator supports the **handleInvalid** parameter to choose how to handle invalid input during transforming data. Available options include ‘keep’ (any invalid inputs are assigned to an extra categorical index) and ‘error’ (throw an error).

OneHotEncoderEstimator

Input DataFrame

```
+-------+-----------+
|Weather|Temperature|
+-------+-----------+
|    Fog|         30|
|   Rain|         25|
|    Sun|         36|
+-------+-----------+
```

OneHotEncoderEstimator

```
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame([
    ('Fog', 30),
    ('Rain', 25),
    ('Sun',36),
], ["Weather", "Temperature"])

indexer = StringIndexer(inputCol="Weather", outputCol="WeatherIndex")

indexerModel = indexer.fit(df)
indexedDF=indexerModel.transform(df)

encoder = OneHotEncoderEstimator(inputCols=["WeatherIndex"],
                                 outputCols=["WeatherOneHot"])
model = encoder.fit(indexedDF)
encodedDF = model.transform(indexedDF)
```

OneHotEncoderEstimator

```
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame([
    ('Fog', 30),
    ('Rain', 25),
    ('Sun',36),
], ["Weather", "Temperature"])

indexer = StringIndexer(inputCol="Weather", outputCol="WeatherIndex")

indexerModel = indexer.fit(df)
indexedDF=indexerModel.transform(df)

encoder = OneHotEncoderEstimator(inputCols=["WeatherIndex"],
                                 outputCols=["WeatherOneHot"])
model = encoder.fit(indexedDF)
encodedDF = model.transform(indexedDF)
```

First we transform “Weather” into an index “WeatherIndex”

OneHotEncoderEstimator

```
from pyspark.ml.feature import OneHotEncoderEstimator
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame([
    ('Fog', 30),
    ('Rain', 25),
    ('Sun',36),
], ["Weather", "Temperature"])

indexer = StringIndexer(inputCol="Weather", outputCol="WeatherIndex")

indexerModel = indexer.fit(df)
indexedDF=indexerModel.transform(df)

encoder = OneHotEncoderEstimator(inputCols=["WeatherIndex"],
                                 outputCols=["WeatherOneHot"])
model = encoder.fit(indexedDF)
encodedDF = model.transform(indexedDF)
```

Then we transform the index into an output vectors of zero and one-values

OneHotEncoderEstimator

Input DataFrame

```
+-------+-----------+
|Weather|Temperature|
+-------+-----------+
|    Fog|         30|
|   Rain|         25|
|    Sun|         36|
+-------+-----------+
```

Output vectors are saved as SparseVectors of zeros and ones (the empty vector is also present)

Transformed DataFrame

```
+-------+-----------+------------+-------------+
|Weather|Temperature|WeatherIndex|WeatherOneHot|
+-------+-----------+------------+-------------+
|    Fog|         30|         2.0|    (2,[],[])|
|   Rain|         25|         1.0|(2,[1],[1.0])|
|    Sun|         36|         0.0|(2,[0],[1.0])|
+-------+-----------+------------+-------------+
```