**Intro**

Pipeline concept is definitely not new for software world, Unix pipe operator (|) links two tasks putting the output of one as the input of the other. In machine learning solutions it is pretty much usual to apply several transformation and manipulation to datasets, or to different portions or sample of the same dataset (from classic test/train slices to samples taken for cross-validation procedure). In these cases, pipelines are definitely useful to define a sequence of tasks chained together to define a complete process, which in turn simplifies the operation of the ml solution. In addition, in BigData environment, it is possible to apply the “laziness” of execution to the entire process in order to make it more scalable and robust, therefore no surprise to see pipeline implemented in Spark machine learning library and R API available, by SparklyR package, to leverage the construct. Pipeline component in Spark are basically of two types :

* **transformer:** since dataframe usually need to undergo several kinds of changes column-wide, row-wide or even single value-wide, transformers are the component meant to deliver these transformations. Typically a transformer has a table or dataframe as input and delivers a table/dataframe as output. Sparks, through MLlib, provide a set of feature’s transformers meant to address most common transformations needed;

Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus. Denote a term by tt, a document by dd, and the corpus by DD. Term frequency TF(t,d)TF(t,d) is the number of times that term tt appears in document dd, while document frequency DF(t,D)DF(t,D) is the number of documents that contains term tt. If we only use term frequency to measure the importance, it is very easy to over-emphasize terms that appear very often but carry little information about the document, e.g. “a”, “the”, and “of”. If a term appears very often across the corpus, it means it doesn’t carry special information about a particular document. Inverse document frequency is a numerical measure of how much information a term provides:

IDF(t,D)=log|D|+1DF(t,D)+1,IDF(t,D)=log⁡|D|+1DF(t,D)+1,

where |D||D| is the total number of documents in the corpus. Since logarithm is used, if a term appears in all documents, its IDF value becomes 0. Note that a smoothing term is applied to avoid dividing by zero for terms outside the corpus. The TF-IDF measure is simply the product of TF and IDF:

TFIDF(t,d,D)=TF(t,d)⋅IDF(t,D).TFIDF(t,d,D)=TF(t,d)⋅IDF(t,D).

There are several variants on the definition of term frequency and document frequency. In MLlib, we separate TF and IDF to make them flexible.

**TF**: Both HashingTF and CountVectorizer can be used to generate the term frequency vectors.

HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. In text processing, a “set of terms” might be a bag of words. HashingTF utilizes the hashing trick. A raw feature is mapped into an index (term) by applying a hash function. The hash function used here is MurmurHash 3. Then term frequencies are calculated based on the mapped indices. This approach avoids the need to compute a global term-to-index map, which can be expensive for a large corpus, but it suffers from potential hash collisions, where different raw features may become the same term after hashing. To reduce the chance of collision, we can increase the target feature dimension, i.e. the number of buckets of the hash table. Since a simple modulo on the hashed value is used to determine the vector index, it is advisable to use a power of two as the feature dimension, otherwise the features will not be mapped evenly to the vector indices. The default feature dimension is 218=262,144218=262,144. An optional binary toggle parameter controls term frequency counts. When set to true all nonzero frequency counts are set to 1. This is especially useful for discrete probabilistic models that model binary, rather than integer, counts.

**IDF**: IDF is an Estimator which is fit on a dataset and produces an IDFModel. The IDFModel takes feature vectors (generally created from HashingTF or CountVectorizer) and scales each feature. Intuitively, it down-weights features which appear frequently in a corpus.

**Examples**

In the following code segment, we start with a set of sentences. We split each sentence into words using Tokenizer. For each sentence (bag of words), we use HashingTF to hash the sentence into a feature vector. We use IDF to rescale the feature vectors; this generally improves performance when using text as features. Our feature vectors could then be passed to a learning algorithm.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.**{**HashingTF**, **IDF**, **Tokenizer**}

**val** sentenceData **=** spark.createDataFrame(**Seq**(

(0.0, "Hi I heard about Spark"),

(0.0, "I wish Java could use case classes"),

(1.0, "Logistic regression models are neat")

)).toDF("label", "sentence")

**val** tokenizer **=** **new** **Tokenizer**().setInputCol("sentence").setOutputCol("words")

**val** wordsData **=** tokenizer.transform(sentenceData)

**val** hashingTF **=** **new** **HashingTF**()

.setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)

**val** featurizedData **=** hashingTF.transform(wordsData)

*// alternatively, CountVectorizer can also be used to get term frequency vectors*

**val** idf **=** **new** **IDF**().setInputCol("rawFeatures").setOutputCol("features")

**val** idfModel **=** idf.fit(featurizedData)

**val** rescaledData **=** idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/TfIdfExample.scala" in the Spark repo.

**Word2Vec**

Word2Vec is an Estimator which takes sequences of words representing documents and trains a Word2VecModel. The model maps each word to a unique fixed-size vector. The Word2VecModel transforms each document into a vector using the average of all words in the document; this vector can then be used as features for prediction, document similarity calculations, etc. Please refer to the MLlib user guide on Word2Vec for more details.

**Examples**

In the following code segment, we start with a set of documents, each of which is represented as a sequence of words. For each document, we transform it into a feature vector. This feature vector could then be passed to a learning algorithm.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Word2Vec**

**import** **org.apache.spark.ml.linalg.Vector**

**import** **org.apache.spark.sql.Row**

*// Input data: Each row is a bag of words from a sentence or document.*

**val** documentDF **=** spark.createDataFrame(**Seq**(

"Hi I heard about Spark".split(" "),

"I wish Java could use case classes".split(" "),

"Logistic regression models are neat".split(" ")

).map(Tuple1.apply)).toDF("text")

*// Learn a mapping from words to Vectors.*

**val** word2Vec **=** **new** **Word2Vec**()

.setInputCol("text")

.setOutputCol("result")

.setVectorSize(3)

.setMinCount(0)

**val** model **=** word2Vec.fit(documentDF)

**val** result **=** model.transform(documentDF)

result.collect().foreach { **case** **Row**(text**:** Seq[**\_**], features**:** Vector) **=>**

println(s"Text: [${text.mkString(", ")}] => \nVector: $features\n") }

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/Word2VecExample.scala" in the Spark repo.

**CountVectorizer**

CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

During the fitting process, CountVectorizer will select the top vocabSize words ordered by term frequency across the corpus. An optional parameter minDF also affects the fitting process by specifying the minimum number (or fraction if < 1.0) of documents a term must appear in to be included in the vocabulary. Another optional binary toggle parameter controls the output vector. If set to true all nonzero counts are set to 1. This is especially useful for discrete probabilistic models that model binary, rather than integer, counts.

**Examples**

Assume that we have the following DataFrame with columns id and texts:

id | texts

----|----------

0 | Array("a", "b", "c")

1 | Array("a", "b", "b", "c", "a")

each row in texts is a document of type Array[String]. Invoking fit of CountVectorizer produces a CountVectorizerModel with vocabulary (a, b, c). Then the output column “vector” after transformation contains:

id | texts | vector

----|---------------------------------|---------------

0 | Array("a", "b", "c") | (3,[0,1,2],[1.0,1.0,1.0])

1 | Array("a", "b", "b", "c", "a") | (3,[0,1,2],[2.0,2.0,1.0])

Each vector represents the token counts of the document over the vocabulary.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.**{**CountVectorizer**, **CountVectorizerModel**}

**val** df **=** spark.createDataFrame(**Seq**(

(0, **Array**("a", "b", "c")),

(1, **Array**("a", "b", "b", "c", "a"))

)).toDF("id", "words")

*// fit a CountVectorizerModel from the corpus*

**val** cvModel**:** CountVectorizerModel = **new** **CountVectorizer**()

.setInputCol("words")

.setOutputCol("features")

.setVocabSize(3)

.setMinDF(2)

.fit(df)

*// alternatively, define CountVectorizerModel with a-priori vocabulary*

**val** cvm **=** **new** **CountVectorizerModel**(**Array**("a", "b", "c"))

.setInputCol("words")

.setOutputCol("features")

cvModel.transform(df).show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/CountVectorizerExample.scala" in the Spark repo.

**FeatureHasher**

Feature hashing projects a set of categorical or numerical features into a feature vector of specified dimension (typically substantially smaller than that of the original feature space). This is done using the hashing trick to map features to indices in the feature vector.

The FeatureHasher transformer operates on multiple columns. Each column may contain either numeric or categorical features. Behavior and handling of column data types is as follows:

* Numeric columns: For numeric features, the hash value of the column name is used to map the feature value to its index in the feature vector. By default, numeric features are not treated as categorical (even when they are integers). To treat them as categorical, specify the relevant columns using the categoricalCols parameter.
* String columns: For categorical features, the hash value of the string “column\_name=value” is used to map to the vector index, with an indicator value of 1.0. Thus, categorical features are “one-hot” encoded (similarly to using OneHotEncoder with dropLast=false).
* Boolean columns: Boolean values are treated in the same way as string columns. That is, boolean features are represented as “column\_name=true” or “column\_name=false”, with an indicator value of 1.0.

Null (missing) values are ignored (implicitly zero in the resulting feature vector).

The hash function used here is also the MurmurHash 3 used in HashingTF. Since a simple modulo on the hashed value is used to determine the vector index, it is advisable to use a power of two as the numFeatures parameter; otherwise the features will not be mapped evenly to the vector indices.

**Examples**

Assume that we have a DataFrame with 4 input columns real, bool, stringNum, and string. These different data types as input will illustrate the behavior of the transform to produce a column of feature vectors.

real| bool|stringNum|string

----|-----|---------|------

2.2| true| 1| foo

3.3|false| 2| bar

4.4|false| 3| baz

5.5|false| 4| foo

Then the output of FeatureHasher.transform on this DataFrame is:

real|bool |stringNum|string|features

----|-----|---------|------|-------------------------------------------------------

2.2 |true |1 |foo |(262144,[51871, 63643,174475,253195],[1.0,1.0,2.2,1.0])

3.3 |false|2 |bar |(262144,[6031, 80619,140467,174475],[1.0,1.0,1.0,3.3])

4.4 |false|3 |baz |(262144,[24279,140467,174475,196810],[1.0,1.0,4.4,1.0])

5.5 |false|4 |foo |(262144,[63643,140467,168512,174475],[1.0,1.0,1.0,5.5])

The resulting feature vectors could then be passed to a learning algorithm.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.FeatureHasher**

**val** dataset **=** spark.createDataFrame(**Seq**(

(2.2, **true**, "1", "foo"),

(3.3, **false**, "2", "bar"),

(4.4, **false**, "3", "baz"),

(5.5, **false**, "4", "foo")

)).toDF("real", "bool", "stringNum", "string")

**val** hasher **=** **new** **FeatureHasher**()

.setInputCols("real", "bool", "stringNum", "string")

.setOutputCol("features")

**val** featurized **=** hasher.transform(dataset)

featurized.show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/FeatureHasherExample.scala" in the Spark repo.

**Feature Transformers**

**Tokenizer**

Tokenization is the process of taking text (such as a sentence) and breaking it into individual terms (usually words). A simple Tokenizer class provides this functionality. The example below shows how to split sentences into sequences of words.

RegexTokenizer allows more advanced tokenization based on regular expression (regex) matching. By default, the parameter “pattern” (regex, default: "\\s+") is used as delimiters to split the input text. Alternatively, users can set parameter “gaps” to false indicating the regex “pattern” denotes “tokens” rather than splitting gaps, and find all matching occurrences as the tokenization result.

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.**{**RegexTokenizer**, **Tokenizer**}

**import** **org.apache.spark.sql.SparkSession**

**import** **org.apache.spark.sql.functions.\_**

**val** sentenceDataFrame **=** spark.createDataFrame(**Seq**(

(0, "Hi I heard about Spark"),

(1, "I wish Java could use case classes"),

(2, "Logistic,regression,models,are,neat")

)).toDF("id", "sentence")

**val** tokenizer **=** **new** **Tokenizer**().setInputCol("sentence").setOutputCol("words")

**val** regexTokenizer **=** **new** **RegexTokenizer**()

.setInputCol("sentence")

.setOutputCol("words")

.setPattern("\\W") *// alternatively .setPattern("\\w+").setGaps(false)*

**val** countTokens **=** udf { (words**:** Seq[String]) **=>** words.length }

**val** tokenized **=** tokenizer.transform(sentenceDataFrame)

tokenized.select("sentence", "words")

.withColumn("tokens", countTokens(col("words"))).show(**false**)

**val** regexTokenized **=** regexTokenizer.transform(sentenceDataFrame)

regexTokenized.select("sentence", "words")

.withColumn("tokens", countTokens(col("words"))).show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/TokenizerExample.scala" in the Spark repo.

**StopWordsRemover**

Stop words are words which should be excluded from the input, typically because the words appear frequently and don’t carry as much meaning.

StopWordsRemover takes as input a sequence of strings (e.g. the output of a Tokenizer) and drops all the stop words from the input sequences. The list of stopwords is specified by the stopWords parameter. Default stop words for some languages are accessible by calling StopWordsRemover.loadDefaultStopWords(language), for which available options are “danish”, “dutch”, “english”, “finnish”, “french”, “german”, “hungarian”, “italian”, “norwegian”, “portuguese”, “russian”, “spanish”, “swedish” and “turkish”. A boolean parameter caseSensitive indicates if the matches should be case sensitive (false by default).

**Examples**

Assume that we have the following DataFrame with columns id and raw:

id | raw

----|----------

0 | [I, saw, the, red, balloon]

1 | [Mary, had, a, little, lamb]

Applying StopWordsRemover with raw as the input column and filtered as the output column, we should get the following:

id | raw | filtered

----|-----------------------------|--------------------

0 | [I, saw, the, red, balloon] | [saw, red, balloon]

1 | [Mary, had, a, little, lamb]|[Mary, little, lamb]

In filtered, the stop words “I”, “the”, “had”, and “a” have been filtered out.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.StopWordsRemover**

**val** remover **=** **new** **StopWordsRemover**()

.setInputCol("raw")

.setOutputCol("filtered")

**val** dataSet **=** spark.createDataFrame(**Seq**(

(0, **Seq**("I", "saw", "the", "red", "balloon")),

(1, **Seq**("Mary", "had", "a", "little", "lamb"))

)).toDF("id", "raw")

remover.transform(dataSet).show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/StopWordsRemoverExample.scala" in the Spark repo.

nn**-gram**

An n-gram is a sequence of nn tokens (typically words) for some integer nn. The NGram class can be used to transform input features into nn-grams.

NGram takes as input a sequence of strings (e.g. the output of a Tokenizer). The parameter n is used to determine the number of terms in each nn-gram. The output will consist of a sequence of nn-grams where each nn-gram is represented by a space-delimited string of nn consecutive words. If the input sequence contains fewer than n strings, no output is produced.

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.NGram**

**val** wordDataFrame **=** spark.createDataFrame(**Seq**(

(0, **Array**("Hi", "I", "heard", "about", "Spark")),

(1, **Array**("I", "wish", "Java", "could", "use", "case", "classes")),

(2, **Array**("Logistic", "regression", "models", "are", "neat"))

)).toDF("id", "words")

**val** ngram **=** **new** **NGram**().setN(2).setInputCol("words").setOutputCol("ngrams")

**val** ngramDataFrame **=** ngram.transform(wordDataFrame)

ngramDataFrame.select("ngrams").show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/NGramExample.scala" in the Spark repo.

**Binarizer**

Binarization is the process of thresholding numerical features to binary (0/1) features.

Binarizer takes the common parameters inputCol and outputCol, as well as the threshold for binarization. Feature values greater than the threshold are binarized to 1.0; values equal to or less than the threshold are binarized to 0.0. Both Vector and Double types are supported for inputCol.

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Binarizer**

**val** data **=** **Array**((0, 0.1), (1, 0.8), (2, 0.2))

**val** dataFrame **=** spark.createDataFrame(data).toDF("id", "feature")

**val** binarizer**:** Binarizer = **new** **Binarizer**()

.setInputCol("feature")

.setOutputCol("binarized\_feature")

.setThreshold(0.5)

**val** binarizedDataFrame **=** binarizer.transform(dataFrame)

println(s"Binarizer output with Threshold = ${binarizer.getThreshold}")

binarizedDataFrame.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/BinarizerExample.scala" in the Spark repo.

**PCA**

PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. A PCA class trains a model to project vectors to a low-dimensional space using PCA. The example below shows how to project 5-dimensional feature vectors into 3-dimensional principal components.

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.PCA**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Array**(

Vectors.sparse(5, **Seq**((1, 1.0), (3, 7.0))),

Vectors.dense(2.0, 0.0, 3.0, 4.0, 5.0),

Vectors.dense(4.0, 0.0, 0.0, 6.0, 7.0)

)

**val** df **=** spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

**val** pca **=** **new** **PCA**()

.setInputCol("features")

.setOutputCol("pcaFeatures")

.setK(3)

.fit(df)

**val** result **=** pca.transform(df).select("pcaFeatures")

result.show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/PCAExample.scala" in the Spark repo.

**PolynomialExpansion**

Polynomial expansion is the process of expanding your features into a polynomial space, which is formulated by an n-degree combination of original dimensions. A PolynomialExpansion class provides this functionality. The example below shows how to expand your features into a 3-degree polynomial space.

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.PolynomialExpansion**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Array**(

Vectors.dense(2.0, 1.0),

Vectors.dense(0.0, 0.0),

Vectors.dense(3.0, -1.0)

)

**val** df **=** spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

**val** polyExpansion **=** **new** **PolynomialExpansion**()

.setInputCol("features")

.setOutputCol("polyFeatures")

.setDegree(3)

**val** polyDF **=** polyExpansion.transform(df)

polyDF.show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/PolynomialExpansionExample.scala" in the Spark repo.

**Discrete Cosine Transform (DCT)**

The Discrete Cosine Transform transforms a length NN real-valued sequence in the time domain into another length NN real-valued sequence in the frequency domain. A DCT class provides this functionality, implementing the DCT-II and scaling the result by 1/2–√1/2 such that the representing matrix for the transform is unitary. No shift is applied to the transformed sequence (e.g. the 00th element of the transformed sequence is the 00th DCT coefficient and *not* the N/2N/2th).

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.DCT**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Seq**(

Vectors.dense(0.0, 1.0, -2.0, 3.0),

Vectors.dense(-1.0, 2.0, 4.0, -7.0),

Vectors.dense(14.0, -2.0, -5.0, 1.0))

**val** df **=** spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

**val** dct **=** **new** **DCT**()

.setInputCol("features")

.setOutputCol("featuresDCT")

.setInverse(**false**)

**val** dctDf **=** dct.transform(df)

dctDf.select("featuresDCT").show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/DCTExample.scala" in the Spark repo.

**StringIndexer**

StringIndexer encodes a string column of labels to a column of label indices. StringIndexer can encode multiple columns. The indices are in [0, numLabels), and four ordering options are supported: “frequencyDesc”: descending order by label frequency (most frequent label assigned 0), “frequencyAsc”: ascending order by label frequency (least frequent label assigned 0), “alphabetDesc”: descending alphabetical order, and “alphabetAsc”: ascending alphabetical order (default = “frequencyDesc”). Note that in case of equal frequency when under “frequencyDesc”/”frequencyAsc”, the strings are further sorted by alphabet.

The unseen labels will be put at index numLabels if user chooses to keep them. If the input column is numeric, we cast it to string and index the string values. When downstream pipeline components such as Estimator or Transformer make use of this string-indexed label, you must set the input column of the component to this string-indexed column name. In many cases, you can set the input column with setInputCol.

**Examples**

Assume that we have the following DataFrame with columns id and category:

id | category

----|----------

0 | a

1 | b

2 | c

3 | a

4 | a

5 | c

category is a string column with three labels: “a”, “b”, and “c”. Applying StringIndexer with category as the input column and categoryIndex as the output column, we should get the following:

id | category | categoryIndex

----|----------|---------------

0 | a | 0.0

1 | b | 2.0

2 | c | 1.0

3 | a | 0.0

4 | a | 0.0

5 | c | 1.0

“a” gets index 0 because it is the most frequent, followed by “c” with index 1 and “b” with index 2.

Additionally, there are three strategies regarding how StringIndexer will handle unseen labels when you have fit a StringIndexer on one dataset and then use it to transform another:

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

**Examples**

Let’s go back to our previous example but this time reuse our previously defined StringIndexer on the following dataset:

id | category

----|----------

0 | a

1 | b

2 | c

3 | d

4 | e

If you’ve not set how StringIndexer handles unseen labels or set it to “error”, an exception will be thrown. However, if you had called setHandleInvalid("skip"), the following dataset will be generated:

id | category | categoryIndex

----|----------|---------------

0 | a | 0.0

1 | b | 2.0

2 | c | 1.0

Notice that the rows containing “d” or “e” do not appear.

If you call setHandleInvalid("keep"), the following dataset will be generated:

id | category | categoryIndex

----|----------|---------------

0 | a | 0.0

1 | b | 2.0

2 | c | 1.0

3 | d | 3.0

4 | e | 3.0

Notice that the rows containing “d” or “e” are mapped to index “3.0”

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.StringIndexer**

**val** df **=** spark.createDataFrame(

**Seq**((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))

).toDF("id", "category")

**val** indexer **=** **new** **StringIndexer**()

.setInputCol("category")

.setOutputCol("categoryIndex")

**val** indexed **=** indexer.fit(df).transform(df)

indexed.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/StringIndexerExample.scala" in the Spark repo.

**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 with IndexToString. However, you are free to supply your own labels.

**Examples**

Building on the StringIndexer example, let’s assume we have the following DataFrame with columns id and categoryIndex:

id | categoryIndex

----|---------------

0 | 0.0

1 | 2.0

2 | 1.0

3 | 0.0

4 | 0.0

5 | 1.0

Applying IndexToString with categoryIndex as the input column, originalCategory as the output column, we are able to retrieve our original labels (they will be inferred from the columns’ metadata):

id | categoryIndex | originalCategory

----|---------------|-----------------

0 | 0.0 | a

1 | 2.0 | b

2 | 1.0 | c

3 | 0.0 | a

4 | 0.0 | a

5 | 1.0 | c

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.attribute.Attribute**

**import** **org.apache.spark.ml.feature.**{**IndexToString**, **StringIndexer**}

**val** df **=** spark.createDataFrame(**Seq**(

(0, "a"),

(1, "b"),

(2, "c"),

(3, "a"),

(4, "a"),

(5, "c")

)).toDF("id", "category")

**val** indexer **=** **new** **StringIndexer**()

.setInputCol("category")

.setOutputCol("categoryIndex")

.fit(df)

**val** indexed **=** indexer.transform(df)

println(s"Transformed string column '${indexer.getInputCol}' " +

s"to indexed column '${indexer.getOutputCol}'")

indexed.show()

**val** inputColSchema **=** indexed.schema(indexer.getOutputCol)

println(s"StringIndexer will store labels in output column metadata: " +

s"${Attribute.fromStructField(inputColSchema).toString}\n")

**val** converter **=** **new** **IndexToString**()

.setInputCol("categoryIndex")

.setOutputCol("originalCategory")

**val** converted **=** converter.transform(indexed)

println(s"Transformed indexed column '${converter.getInputCol}' back to original string " +

s"column '${converter.getOutputCol}' using labels in metadata")

converted.select("id", "categoryIndex", "originalCategory").show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/IndexToStringExample.scala" in the Spark repo.

**OneHotEncoder**

One-hot encoding maps a categorical feature, represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features. For string type input data, it is common to encode categorical features using StringIndexer first.

OneHotEncoder 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.

OneHotEncoder 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).

**Examples**

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.OneHotEncoder**

**val** df **=** spark.createDataFrame(**Seq**(

(0.0, 1.0),

(1.0, 0.0),

(2.0, 1.0),

(0.0, 2.0),

(0.0, 1.0),

(2.0, 0.0)

)).toDF("categoryIndex1", "categoryIndex2")

**val** encoder **=** **new** **OneHotEncoder**()

.setInputCols(**Array**("categoryIndex1", "categoryIndex2"))

.setOutputCols(**Array**("categoryVec1", "categoryVec2"))

**val** model **=** encoder.fit(df)

**val** encoded **=** model.transform(df)

encoded.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/OneHotEncoderExample.scala" in the Spark repo.

**VectorIndexer**

VectorIndexer helps index categorical features in datasets of Vectors. It can both automatically decide which features are categorical and convert original values to category indices. Specifically, it does the following:

1. Take an input column of type Vector and a parameter maxCategories.
2. Decide which features should be categorical based on the number of distinct values, where features with at most maxCategories are declared categorical.
3. Compute 0-based category indices for each categorical feature.
4. Index categorical features and transform original feature values to indices.

Indexing categorical features allows algorithms such as Decision Trees and Tree Ensembles to treat categorical features appropriately, improving performance.

**Examples**

In the example below, we read in a dataset of labeled points and then use VectorIndexer to decide which features should be treated as categorical. We transform the categorical feature values to their indices. This transformed data could then be passed to algorithms such as DecisionTreeRegressor that handle categorical features.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.VectorIndexer**

**val** data **=** spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")

**val** indexer **=** **new** **VectorIndexer**()

.setInputCol("features")

.setOutputCol("indexed")

.setMaxCategories(10)

**val** indexerModel **=** indexer.fit(data)

**val** categoricalFeatures**:** Set[Int] **=** indexerModel.categoryMaps.keys.toSet

println(s"Chose ${categoricalFeatures.size} " +

s"categorical features: ${categoricalFeatures.mkString(", ")}")

*// Create new column "indexed" with categorical values transformed to indices*

**val** indexedData **=** indexerModel.transform(data)

indexedData.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/VectorIndexerExample.scala" in the Spark repo.

**Interaction**

Interaction is a Transformer which takes vector or double-valued columns, and generates a single vector column that contains the product of all combinations of one value from each input column.

For example, if you have 2 vector type columns each of which has 3 dimensions as input columns, then you’ll get a 9-dimensional vector as the output column.

**Examples**

Assume that we have the following DataFrame with the columns “id1”, “vec1”, and “vec2”:

id1|vec1 |vec2

---|--------------|--------------

1 |[1.0,2.0,3.0] |[8.0,4.0,5.0]

2 |[4.0,3.0,8.0] |[7.0,9.0,8.0]

3 |[6.0,1.0,9.0] |[2.0,3.0,6.0]

4 |[10.0,8.0,6.0]|[9.0,4.0,5.0]

5 |[9.0,2.0,7.0] |[10.0,7.0,3.0]

6 |[1.0,1.0,4.0] |[2.0,8.0,4.0]

Applying Interaction with those input columns, then interactedCol as the output column contains:

id1|vec1 |vec2 |interactedCol

---|--------------|--------------|------------------------------------------------------

1 |[1.0,2.0,3.0] |[8.0,4.0,5.0] |[8.0,4.0,5.0,16.0,8.0,10.0,24.0,12.0,15.0]

2 |[4.0,3.0,8.0] |[7.0,9.0,8.0] |[56.0,72.0,64.0,42.0,54.0,48.0,112.0,144.0,128.0]

3 |[6.0,1.0,9.0] |[2.0,3.0,6.0] |[36.0,54.0,108.0,6.0,9.0,18.0,54.0,81.0,162.0]

4 |[10.0,8.0,6.0]|[9.0,4.0,5.0] |[360.0,160.0,200.0,288.0,128.0,160.0,216.0,96.0,120.0]

5 |[9.0,2.0,7.0] |[10.0,7.0,3.0]|[450.0,315.0,135.0,100.0,70.0,30.0,350.0,245.0,105.0]

6 |[1.0,1.0,4.0] |[2.0,8.0,4.0] |[12.0,48.0,24.0,12.0,48.0,24.0,48.0,192.0,96.0]

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Interaction**

**import** **org.apache.spark.ml.feature.VectorAssembler**

**val** df **=** spark.createDataFrame(**Seq**(

(1, 1, 2, 3, 8, 4, 5),

(2, 4, 3, 8, 7, 9, 8),

(3, 6, 1, 9, 2, 3, 6),

(4, 10, 8, 6, 9, 4, 5),

(5, 9, 2, 7, 10, 7, 3),

(6, 1, 1, 4, 2, 8, 4)

)).toDF("id1", "id2", "id3", "id4", "id5", "id6", "id7")

**val** assembler1 **=** **new** **VectorAssembler**().

setInputCols(**Array**("id2", "id3", "id4")).

setOutputCol("vec1")

**val** assembled1 **=** assembler1.transform(df)

**val** assembler2 **=** **new** **VectorAssembler**().

setInputCols(**Array**("id5", "id6", "id7")).

setOutputCol("vec2")

**val** assembled2 **=** assembler2.transform(assembled1).select("id1", "vec1", "vec2")

**val** interaction **=** **new** **Interaction**()

.setInputCols(**Array**("id1", "vec1", "vec2"))

.setOutputCol("interactedCol")

**val** interacted **=** interaction.transform(assembled2)

interacted.show(truncate **=** **false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/InteractionExample.scala" in the Spark repo.

**Normalizer**

Normalizer is a Transformer which transforms a dataset of Vector rows, normalizing each Vector to have unit norm. It takes parameter p, which specifies the p-norm used for normalization. (p=2p=2 by default.) This normalization can help standardize your input data and improve the behavior of learning algorithms.

**Examples**

The following example demonstrates how to load a dataset in libsvm format and then normalize each row to have unit L1L1 norm and unit L∞L∞ norm.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Normalizer**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** dataFrame **=** spark.createDataFrame(**Seq**(

(0, Vectors.dense(1.0, 0.5, -1.0)),

(1, Vectors.dense(2.0, 1.0, 1.0)),

(2, Vectors.dense(4.0, 10.0, 2.0))

)).toDF("id", "features")

*// Normalize each Vector using $L^1$ norm.*

**val** normalizer **=** **new** **Normalizer**()

.setInputCol("features")

.setOutputCol("normFeatures")

.setP(1.0)

**val** l1NormData **=** normalizer.transform(dataFrame)

println("Normalized using L^1 norm")

l1NormData.show()

*// Normalize each Vector using $L^\infty$ norm.*

**val** lInfNormData **=** normalizer.transform(dataFrame, normalizer.p -> Double.PositiveInfinity)

println("Normalized using L^inf norm")

lInfNormData.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/NormalizerExample.scala" in the Spark repo.

**StandardScaler**

StandardScaler transforms a dataset of Vector rows, normalizing each feature to have unit standard deviation and/or zero mean. 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. It will build a dense output, so take care when applying to sparse input.

StandardScaler is an Estimator which can be fit on a dataset to produce a StandardScalerModel; this amounts to computing summary statistics. The model can then transform a Vector column in a dataset to have unit standard deviation and/or zero mean features.

Note that if the standard deviation of a feature is zero, it will return default 0.0 value in the Vector for that feature.

**Examples**

The following example demonstrates how to load a dataset in libsvm format and then normalize each feature to have unit standard deviation.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.StandardScaler**

**val** dataFrame **=** spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")

**val** scaler **=** **new** **StandardScaler**()

.setInputCol("features")

.setOutputCol("scaledFeatures")

.setWithStd(**true**)

.setWithMean(**false**)

*// Compute summary statistics by fitting the StandardScaler.*

**val** scalerModel **=** scaler.fit(dataFrame)

*// Normalize each feature to have unit standard deviation.*

**val** scaledData **=** scalerModel.transform(dataFrame)

scaledData.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/StandardScalerExample.scala" in the Spark repo.

**RobustScaler**

RobustScaler transforms a dataset of Vector rows, removing the median and scaling the data according to a specific quantile range (by default the IQR: Interquartile Range, quantile range between the 1st quartile and the 3rd quartile). Its behavior is quite similar to StandardScaler, however the median and the quantile range are used instead of mean and standard deviation, which make it robust to outliers. It takes parameters:

* lower: 0.25 by default. Lower quantile to calculate quantile range, shared by all features.
* upper: 0.75 by default. Upper quantile to calculate quantile range, shared by all features.
* withScaling: True by default. Scales the data to quantile range.
* withCentering: False by default. Centers the data with median before scaling. It will build a dense output, so take care when applying to sparse input.

RobustScaler is an Estimator which can be fit on a dataset to produce a RobustScalerModel; this amounts to computing quantile statistics. The model can then transform a Vector column in a dataset to have unit quantile range and/or zero median features.

Note that if the quantile range of a feature is zero, it will return default 0.0 value in the Vector for that feature.

**Examples**

The following example demonstrates how to load a dataset in libsvm format and then normalize each feature to have unit quantile range.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.RobustScaler**

**val** dataFrame **=** spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")

**val** scaler **=** **new** **RobustScaler**()

.setInputCol("features")

.setOutputCol("scaledFeatures")

.setWithScaling(**true**)

.setWithCentering(**false**)

.setLower(0.25)

.setUpper(0.75)

*// Compute summary statistics by fitting the RobustScaler.*

**val** scalerModel **=** scaler.fit(dataFrame)

*// Transform each feature to have unit quantile range.*

**val** scaledData **=** scalerModel.transform(dataFrame)

scaledData.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/RobustScalerExample.scala" in the Spark repo.

**MinMaxScaler**

MinMaxScaler transforms a dataset of Vector rows, rescaling each feature to a specific range (often [0, 1]). It takes parameters:

* min: 0.0 by default. Lower bound after transformation, shared by all features.
* max: 1.0 by default. Upper bound after transformation, shared by all features.

MinMaxScaler computes summary statistics on a data set and produces a MinMaxScalerModel. The model can then transform each feature individually such that it is in the given range.

The rescaled value for a feature E is calculated as,

Rescaled(ei)=ei−EminEmax−Emin∗(max−min)+min(1)(1)Rescaled(ei)=ei−EminEmax−Emin∗(max−min)+min

For the case Emax==EminEmax==Emin, Rescaled(ei)=0.5∗(max+min)Rescaled(ei)=0.5∗(max+min)

Note that since zero values will probably be transformed to non-zero values, output of the transformer will be DenseVector even for sparse input.

**Examples**

The following example demonstrates how to load a dataset in libsvm format and then rescale each feature to [0, 1].

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.MinMaxScaler**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** dataFrame **=** spark.createDataFrame(**Seq**(

(0, Vectors.dense(1.0, 0.1, -1.0)),

(1, Vectors.dense(2.0, 1.1, 1.0)),

(2, Vectors.dense(3.0, 10.1, 3.0))

)).toDF("id", "features")

**val** scaler **=** **new** **MinMaxScaler**()

.setInputCol("features")

.setOutputCol("scaledFeatures")

*// Compute summary statistics and generate MinMaxScalerModel*

**val** scalerModel **=** scaler.fit(dataFrame)

*// rescale each feature to range [min, max].*

**val** scaledData **=** scalerModel.transform(dataFrame)

println(s"Features scaled to range: [${scaler.getMin}, ${scaler.getMax}]")

scaledData.select("features", "scaledFeatures").show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/MinMaxScalerExample.scala" in the Spark repo.

**MaxAbsScaler**

MaxAbsScaler transforms a dataset of Vector rows, rescaling each feature to range [-1, 1] by dividing through the maximum absolute value in each feature. It does not shift/center the data, and thus does not destroy any sparsity.

MaxAbsScaler computes summary statistics on a data set and produces a MaxAbsScalerModel. The model can then transform each feature individually to range [-1, 1].

**Examples**

The following example demonstrates how to load a dataset in libsvm format and then rescale each feature to [-1, 1].

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.MaxAbsScaler**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** dataFrame **=** spark.createDataFrame(**Seq**(

(0, Vectors.dense(1.0, 0.1, -8.0)),

(1, Vectors.dense(2.0, 1.0, -4.0)),

(2, Vectors.dense(4.0, 10.0, 8.0))

)).toDF("id", "features")

**val** scaler **=** **new** **MaxAbsScaler**()

.setInputCol("features")

.setOutputCol("scaledFeatures")

*// Compute summary statistics and generate MaxAbsScalerModel*

**val** scalerModel **=** scaler.fit(dataFrame)

*// rescale each feature to range [-1, 1]*

**val** scaledData **=** scalerModel.transform(dataFrame)

scaledData.select("features", "scaledFeatures").show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/MaxAbsScalerExample.scala" in the Spark repo.

**Bucketizer**

Bucketizer transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users. It takes a parameter:

* splits: Parameter for mapping continuous features into buckets. With n+1 splits, there are n buckets. A bucket defined by splits x,y holds values in the range [x,y) except the last bucket, which also includes y. Splits should be strictly increasing. Values at -inf, inf must be explicitly provided to cover all Double values; Otherwise, values outside the splits specified will be treated as errors. Two examples of splits are Array(Double.NegativeInfinity, 0.0, 1.0, Double.PositiveInfinity) and Array(0.0, 1.0, 2.0).

Note that if you have no idea of the upper and lower bounds of the targeted column, you should add Double.NegativeInfinity and Double.PositiveInfinity as the bounds of your splits to prevent a potential out of Bucketizer bounds exception.

Note also that the splits that you provided have to be in strictly increasing order, i.e. s0 < s1 < s2 < ... < sn.

More details can be found in the API docs for Bucketizer.

**Examples**

The following example demonstrates how to bucketize a column of Doubles into another index-wised column.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Bucketizer**

**val** splits **=** **Array**(Double.NegativeInfinity, -0.5, 0.0, 0.5, Double.PositiveInfinity)

**val** data **=** **Array**(-999.9, -0.5, -0.3, 0.0, 0.2, 999.9)

**val** dataFrame **=** spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

**val** bucketizer **=** **new** **Bucketizer**()

.setInputCol("features")

.setOutputCol("bucketedFeatures")

.setSplits(splits)

*// Transform original data into its bucket index.*

**val** bucketedData **=** bucketizer.transform(dataFrame)

println(s"Bucketizer output with ${bucketizer.getSplits.length-1} buckets")

bucketedData.show()

**val** splitsArray **=** **Array**(

**Array**(Double.NegativeInfinity, -0.5, 0.0, 0.5, Double.PositiveInfinity),

**Array**(Double.NegativeInfinity, -0.3, 0.0, 0.3, Double.PositiveInfinity))

**val** data2 **=** **Array**(

(-999.9, -999.9),

(-0.5, -0.2),

(-0.3, -0.1),

(0.0, 0.0),

(0.2, 0.4),

(999.9, 999.9))

**val** dataFrame2 **=** spark.createDataFrame(data2).toDF("features1", "features2")

**val** bucketizer2 **=** **new** **Bucketizer**()

.setInputCols(**Array**("features1", "features2"))

.setOutputCols(**Array**("bucketedFeatures1", "bucketedFeatures2"))

.setSplitsArray(splitsArray)

*// Transform original data into its bucket index.*

**val** bucketedData2 **=** bucketizer2.transform(dataFrame2)

println(s"Bucketizer output with [" +

s"${bucketizer2.getSplitsArray(0).length-1}, " +

s"${bucketizer2.getSplitsArray(1).length-1}] buckets for each input column")

bucketedData2.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/BucketizerExample.scala" in the Spark repo.

**ElementwiseProduct**

ElementwiseProduct multiplies each input vector by a provided “weight” vector, using element-wise multiplication. In other words, it scales each column of the dataset by a scalar multiplier. This represents the Hadamard product between the input vector, v and transforming vector, w, to yield a result vector.

⎛⎝⎜⎜v1⋮vN⎞⎠⎟⎟∘⎛⎝⎜⎜w1⋮wN⎞⎠⎟⎟=⎛⎝⎜⎜v1w1⋮vNwN⎞⎠⎟⎟(v1⋮vN)∘(w1⋮wN)=(v1w1⋮vNwN)

**Examples**

This example below demonstrates how to transform vectors using a transforming vector value.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.ElementwiseProduct**

**import** **org.apache.spark.ml.linalg.Vectors**

*// Create some vector data; also works for sparse vectors*

**val** dataFrame **=** spark.createDataFrame(**Seq**(

("a", Vectors.dense(1.0, 2.0, 3.0)),

("b", Vectors.dense(4.0, 5.0, 6.0)))).toDF("id", "vector")

**val** transformingVector **=** Vectors.dense(0.0, 1.0, 2.0)

**val** transformer **=** **new** **ElementwiseProduct**()

.setScalingVec(transformingVector)

.setInputCol("vector")

.setOutputCol("transformedVector")

*// Batch transform the vectors to create new column:*

transformer.transform(dataFrame).show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/ElementwiseProductExample.scala" in the Spark repo.

**SQLTransformer**

SQLTransformer implements the transformations which are defined by SQL statement. Currently, we only support SQL syntax like "SELECT ... FROM \_\_THIS\_\_ ..." where "\_\_THIS\_\_" represents the underlying table of the input dataset. The select clause specifies the fields, constants, and expressions to display in the output, and can be any select clause that Spark SQL supports. Users can also use Spark SQL built-in function and UDFs to operate on these selected columns. For example, SQLTransformer supports statements like:

* SELECT a, a + b AS a\_b FROM \_\_THIS\_\_
* SELECT a, SQRT(b) AS b\_sqrt FROM \_\_THIS\_\_ where a > 5
* SELECT a, b, SUM(c) AS c\_sum FROM \_\_THIS\_\_ GROUP BY a, b

**Examples**

Assume that we have the following DataFrame with columns id, v1 and v2:

id | v1 | v2

----|-----|-----

0 | 1.0 | 3.0

2 | 2.0 | 5.0

This is the output of the SQLTransformer with statement "SELECT \*, (v1 + v2) AS v3, (v1 \* v2) AS v4 FROM \_\_THIS\_\_":

id | v1 | v2 | v3 | v4

----|-----|-----|-----|-----

0 | 1.0 | 3.0 | 4.0 | 3.0

2 | 2.0 | 5.0 | 7.0 |10.0

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.SQLTransformer**

**val** df **=** spark.createDataFrame(

**Seq**((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")

**val** sqlTrans **=** **new** **SQLTransformer**().setStatement(

"SELECT \*, (v1 + v2) AS v3, (v1 \* v2) AS v4 FROM \_\_THIS\_\_")

sqlTrans.transform(df).show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/SQLTransformerExample.scala" in the Spark repo.

**VectorAssembler**

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

**Examples**

Assume that we have a DataFrame with the columns id, hour, mobile, userFeatures, and clicked:

id | hour | mobile | userFeatures | clicked

----|------|--------|------------------|---------

0 | 18 | 1.0 | [0.0, 10.0, 0.5] | 1.0

userFeatures is a vector column that contains three user features. We want to combine hour, mobile, and userFeatures into a single feature vector called features and use it to predict clicked or not. If we set VectorAssembler’s input columns to hour, mobile, and userFeatures and output column to features, after transformation we should get the following DataFrame:

id | hour | mobile | userFeatures | clicked | features

----|------|--------|------------------|---------|-----------------------------

0 | 18 | 1.0 | [0.0, 10.0, 0.5] | 1.0 | [18.0, 1.0, 0.0, 10.0, 0.5]

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.VectorAssembler**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** dataset **=** spark.createDataFrame(

**Seq**((0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0))

).toDF("id", "hour", "mobile", "userFeatures", "clicked")

**val** assembler **=** **new** **VectorAssembler**()

.setInputCols(**Array**("hour", "mobile", "userFeatures"))

.setOutputCol("features")

**val** output **=** assembler.transform(dataset)

println("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'")

output.select("features", "clicked").show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/VectorAssemblerExample.scala" in the Spark repo.

**VectorSizeHint**

It can sometimes be useful to explicitly specify the size of the vectors for a column of VectorType. For example, VectorAssembler uses size information from its input columns to produce size information and metadata for its output column. While in some cases this information can be obtained by inspecting the contents of the column, in a streaming dataframe the contents are not available until the stream is started. VectorSizeHint allows a user to explicitly specify the vector size for a column so that VectorAssembler, or other transformers that might need to know vector size, can use that column as an input.

To use VectorSizeHint a user must set the inputCol and size parameters. Applying this transformer to a dataframe produces a new dataframe with updated metadata for inputCol specifying the vector size. Downstream operations on the resulting dataframe can get this size using the metadata.

VectorSizeHint can also take an optional handleInvalid parameter which controls its behaviour when the vector column contains nulls or vectors of the wrong size. By default handleInvalid is set to “error”, indicating an exception should be thrown. This parameter can also be set to “skip”, indicating that rows containing invalid values should be filtered out from the resulting dataframe, or “optimistic”, indicating that the column should not be checked for invalid values and all rows should be kept. Note that the use of “optimistic” can cause the resulting dataframe to be in an inconsistent state, meaning the metadata for the column VectorSizeHint was applied to does not match the contents of that column. Users should take care to avoid this kind of inconsistent state.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.**{**VectorAssembler**, **VectorSizeHint**}

**import** **org.apache.spark.ml.linalg.Vectors**

**val** dataset **=** spark.createDataFrame(

**Seq**(

(0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0),

(0, 18, 1.0, Vectors.dense(0.0, 10.0), 0.0))

).toDF("id", "hour", "mobile", "userFeatures", "clicked")

**val** sizeHint **=** **new** **VectorSizeHint**()

.setInputCol("userFeatures")

.setHandleInvalid("skip")

.setSize(3)

**val** datasetWithSize **=** sizeHint.transform(dataset)

println("Rows where 'userFeatures' is not the right size are filtered out")

datasetWithSize.show(**false**)

**val** assembler **=** **new** **VectorAssembler**()

.setInputCols(**Array**("hour", "mobile", "userFeatures"))

.setOutputCol("features")

*// This dataframe can be used by downstream transformers as before*

**val** output **=** assembler.transform(datasetWithSize)

println("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'")

output.select("features", "clicked").show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/VectorSizeHintExample.scala" in the Spark repo.

**QuantileDiscretizer**

QuantileDiscretizer takes a column with continuous features and outputs a column with binned categorical features. The number of bins is set by the numBuckets parameter. It is possible that the number of buckets used will be smaller than this value, for example, if there are too few distinct values of the input to create enough distinct quantiles.

NaN values: NaN values will be removed from the column during QuantileDiscretizer fitting. This will produce a Bucketizer model for making predictions. During the transformation, Bucketizer will raise an error when it finds NaN values in the dataset, but the user can also choose to either keep or remove NaN values within the dataset by setting handleInvalid. If the user chooses to keep NaN values, they will be handled specially and placed into their own bucket, for example, if 4 buckets are used, then non-NaN data will be put into buckets[0-3], but NaNs will be counted in a special bucket[4].

Algorithm: The bin ranges are chosen using an approximate algorithm (see the documentation for approxQuantile for a detailed description). The precision of the approximation can be controlled with the relativeError parameter. When set to zero, exact quantiles are calculated (**Note:** Computing exact quantiles is an expensive operation). The lower and upper bin bounds will be -Infinity and +Infinity covering all real values.

**Examples**

Assume that we have a DataFrame with the columns id, hour:

id | hour

----|------

0 | 18.0

----|------

1 | 19.0

----|------

2 | 8.0

----|------

3 | 5.0

----|------

4 | 2.2

hour is a continuous feature with Double type. We want to turn the continuous feature into a categorical one. Given numBuckets = 3, we should get the following DataFrame:

id | hour | result

----|------|------

0 | 18.0 | 2.0

----|------|------

1 | 19.0 | 2.0

----|------|------

2 | 8.0 | 1.0

----|------|------

3 | 5.0 | 1.0

----|------|------

4 | 2.2 | 0.0

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.QuantileDiscretizer**

**val** data **=** **Array**((0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2))

**val** df **=** spark.createDataFrame(data).toDF("id", "hour")

**val** discretizer **=** **new** **QuantileDiscretizer**()

.setInputCol("hour")

.setOutputCol("result")

.setNumBuckets(3)

**val** result **=** discretizer.fit(df).transform(df)

result.show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/QuantileDiscretizerExample.scala" in the Spark repo.

**Imputer**

The Imputer estimator completes missing values in a dataset, using the mean, median or mode of the columns in which the missing values are located. The input columns should be of numeric type. Currently Imputer does not support categorical features and possibly creates incorrect values for columns containing categorical features. Imputer can impute custom values other than ‘NaN’ by .setMissingValue(custom\_value). For example, .setMissingValue(0) will impute all occurrences of (0).

**Note** all null values in the input columns are treated as missing, and so are also imputed.

**Examples**

Suppose that we have a DataFrame with the columns a and b:

a | b

------------|-----------

1.0 | Double.NaN

2.0 | Double.NaN

Double.NaN | 3.0

4.0 | 4.0

5.0 | 5.0

In this example, Imputer will replace all occurrences of Double.NaN (the default for the missing value) with the mean (the default imputation strategy) computed from the other values in the corresponding columns. In this example, the surrogate values for columns a and b are 3.0 and 4.0 respectively. After transformation, the missing values in the output columns will be replaced by the surrogate value for the relevant column.

a | b | out\_a | out\_b

------------|------------|-------|-------

1.0 | Double.NaN | 1.0 | 4.0

2.0 | Double.NaN | 2.0 | 4.0

Double.NaN | 3.0 | 3.0 | 3.0

4.0 | 4.0 | 4.0 | 4.0

5.0 | 5.0 | 5.0 | 5.0

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.Imputer**

**val** df **=** spark.createDataFrame(**Seq**(

(1.0, Double.NaN),

(2.0, Double.NaN),

(Double.NaN, 3.0),

(4.0, 4.0),

(5.0, 5.0)

)).toDF("a", "b")

**val** imputer **=** **new** **Imputer**()

.setInputCols(**Array**("a", "b"))

.setOutputCols(**Array**("out\_a", "out\_b"))

**val** model **=** imputer.fit(df)

model.transform(df).show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/ImputerExample.scala" in the Spark repo.

**Feature Selectors**

**VectorSlicer**

VectorSlicer is a transformer that takes a feature vector and outputs a new feature vector with a sub-array of the original features. It is useful for extracting features from a vector column.

VectorSlicer accepts a vector column with specified indices, then outputs a new vector column whose values are selected via those indices. There are two types of indices,

1. Integer indices that represent the indices into the vector, setIndices().
2. String indices that represent the names of features into the vector, setNames(). *This requires the vector column to have an AttributeGroup since the implementation matches on the name field of an Attribute.*

Specification by integer and string are both acceptable. Moreover, you can use integer index and string name simultaneously. At least one feature must be selected. Duplicate features are not allowed, so there can be no overlap between selected indices and names. Note that if names of features are selected, an exception will be thrown if empty input attributes are encountered.

The output vector will order features with the selected indices first (in the order given), followed by the selected names (in the order given).

**Examples**

Suppose that we have a DataFrame with the column userFeatures:

userFeatures

------------------

[0.0, 10.0, 0.5]

userFeatures is a vector column that contains three user features. Assume that the first column of userFeatures are all zeros, so we want to remove it and select only the last two columns. The VectorSlicer selects the last two elements with setIndices(1, 2) then produces a new vector column named features:

userFeatures | features

------------------|-----------------------------

[0.0, 10.0, 0.5] | [10.0, 0.5]

Suppose also that we have potential input attributes for the userFeatures, i.e. ["f1", "f2", "f3"], then we can use setNames("f2", "f3") to select them.

userFeatures | features

------------------|-----------------------------

[0.0, 10.0, 0.5] | [10.0, 0.5]

["f1", "f2", "f3"] | ["f2", "f3"]

* **Scala**
* **Java**
* **Python**

**import** **java.util.Arrays**

**import** **org.apache.spark.ml.attribute.**{**Attribute**, **AttributeGroup**, **NumericAttribute**}

**import** **org.apache.spark.ml.feature.VectorSlicer**

**import** **org.apache.spark.ml.linalg.Vectors**

**import** **org.apache.spark.sql.**{**Row**, **SparkSession**}

**import** **org.apache.spark.sql.types.StructType**

**val** data **=** Arrays.asList(

**Row**(Vectors.sparse(3, **Seq**((0, -2.0), (1, 2.3)))),

**Row**(Vectors.dense(-2.0, 2.3, 0.0))

)

**val** defaultAttr **=** NumericAttribute.defaultAttr

**val** attrs **=** **Array**("f1", "f2", "f3").map(defaultAttr.withName)

**val** attrGroup **=** **new** **AttributeGroup**("userFeatures", attrs.asInstanceOf[Array[Attribute]])

**val** dataset **=** spark.createDataFrame(data, **StructType**(**Array**(attrGroup.toStructField())))

**val** slicer **=** **new** **VectorSlicer**().setInputCol("userFeatures").setOutputCol("features")

slicer.setIndices(**Array**(1)).setNames(**Array**("f3"))

*// or slicer.setIndices(Array(1, 2)), or slicer.setNames(Array("f2", "f3"))*

**val** output **=** slicer.transform(dataset)

output.show(**false**)

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/VectorSlicerExample.scala" in the Spark repo.

**RFormula**

RFormula selects columns specified by an R model formula. Currently we support a limited subset of the R operators, including ‘~’, ‘.’, ‘:’, ‘+’, and ‘-‘. The basic operators are:

* ~ separate target and terms
* + concat terms, “+ 0” means removing intercept
* - remove a term, “- 1” means removing intercept
* : interaction (multiplication for numeric values, or binarized categorical values)
* . all columns except target

Suppose a and b are double columns, we use the following simple examples to illustrate the effect of RFormula:

* y ~ a + b means model y ~ w0 + w1 \* a + w2 \* b where w0 is the intercept and w1, w2 are coefficients.
* y ~ a + b + a:b - 1 means model y ~ w1 \* a + w2 \* b + w3 \* a \* b where w1, w2, w3 are coefficients.

RFormula produces a vector column of features and a double or string column of label. Like when formulas are used in R for linear regression, numeric columns will be cast to doubles. As to string input columns, they will first be transformed with StringIndexer using ordering determined by stringOrderType, and the last category after ordering is dropped, then the doubles will be one-hot encoded.

Suppose a string feature column containing values {'b', 'a', 'b', 'a', 'c', 'b'}, we set stringOrderType to control the encoding:

stringOrderType | Category mapped to 0 by StringIndexer | Category dropped by RFormula

----------------|---------------------------------------|---------------------------------

'frequencyDesc' | most frequent category ('b') | least frequent category ('c')

'frequencyAsc' | least frequent category ('c') | most frequent category ('b')

'alphabetDesc' | last alphabetical category ('c') | first alphabetical category ('a')

'alphabetAsc' | first alphabetical category ('a') | last alphabetical category ('c')

If the label column is of type string, it will be first transformed to double with StringIndexer using frequencyDesc ordering. If the label column does not exist in the DataFrame, the output label column will be created from the specified response variable in the formula.

**Note:** The ordering option stringOrderType is NOT used for the label column. When the label column is indexed, it uses the default descending frequency ordering in StringIndexer.

**Examples**

Assume that we have a DataFrame with the columns id, country, hour, and clicked:

id | country | hour | clicked

---|---------|------|---------

7 | "US" | 18 | 1.0

8 | "CA" | 12 | 0.0

9 | "NZ" | 15 | 0.0

If we use RFormula with a formula string of clicked ~ country + hour, which indicates that we want to predict clicked based on country and hour, after transformation we should get the following DataFrame:

id | country | hour | clicked | features | label

---|---------|------|---------|------------------|-------

7 | "US" | 18 | 1.0 | [0.0, 0.0, 18.0] | 1.0

8 | "CA" | 12 | 0.0 | [0.0, 1.0, 12.0] | 0.0

9 | "NZ" | 15 | 0.0 | [1.0, 0.0, 15.0] | 0.0

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.RFormula**

**val** dataset **=** spark.createDataFrame(**Seq**(

(7, "US", 18, 1.0),

(8, "CA", 12, 0.0),

(9, "NZ", 15, 0.0)

)).toDF("id", "country", "hour", "clicked")

**val** formula **=** **new** **RFormula**()

.setFormula("clicked ~ country + hour")

.setFeaturesCol("features")

.setLabelCol("label")

**val** output **=** formula.fit(dataset).transform(dataset)

output.select("features", "label").show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/RFormulaExample.scala" in the Spark repo.

**ChiSqSelector**

ChiSqSelector stands for Chi-Squared feature selection. It operates on labeled data with categorical features. ChiSqSelector uses the Chi-Squared test of independence to decide which features to choose. It supports five selection methods: numTopFeatures, percentile, fpr, fdr, fwe:

* numTopFeatures chooses a fixed number of top features according to a chi-squared test. This is akin to yielding the features with the most predictive power.
* percentile is similar to numTopFeatures but chooses a fraction of all features instead of a fixed number.
* fpr chooses all features whose p-values are below a threshold, thus controlling the false positive rate of selection.
* fdr uses the Benjamini-Hochberg procedure to choose all features whose false discovery rate is below a threshold.
* fwe chooses all features whose p-values are below a threshold. The threshold is scaled by 1/numFeatures, thus controlling the family-wise error rate of selection. By default, the selection method is numTopFeatures, with the default number of top features set to 50. The user can choose a selection method using setSelectorType.

**Examples**

Assume that we have a DataFrame with the columns id, features, and clicked, which is used as our target to be predicted:

id | features | clicked

---|-----------------------|---------

7 | [0.0, 0.0, 18.0, 1.0] | 1.0

8 | [0.0, 1.0, 12.0, 0.0] | 0.0

9 | [1.0, 0.0, 15.0, 0.1] | 0.0

If we use ChiSqSelector with numTopFeatures = 1, then according to our label clicked the last column in our features is chosen as the most useful feature:

id | features | clicked | selectedFeatures

---|-----------------------|---------|------------------

7 | [0.0, 0.0, 18.0, 1.0] | 1.0 | [1.0]

8 | [0.0, 1.0, 12.0, 0.0] | 0.0 | [0.0]

9 | [1.0, 0.0, 15.0, 0.1] | 0.0 | [0.1]

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.ChiSqSelector**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Seq**(

(7, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),

(8, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),

(9, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)

)

**val** df **=** spark.createDataset(data).toDF("id", "features", "clicked")

**val** selector **=** **new** **ChiSqSelector**()

.setNumTopFeatures(1)

.setFeaturesCol("features")

.setLabelCol("clicked")

.setOutputCol("selectedFeatures")

**val** result **=** selector.fit(df).transform(df)

println(s"ChiSqSelector output with top ${selector.getNumTopFeatures} features selected")

result.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/ChiSqSelectorExample.scala" in the Spark repo.

**UnivariateFeatureSelector**

UnivariateFeatureSelector operates on categorical/continuous labels with categorical/continuous features. User can set featureType and labelType, and Spark will pick the score function to use based on the specified featureType and labelType.

featureType | labelType |score function

------------|------------|--------------

categorical |categorical | chi-squared (chi2)

continuous |categorical | ANOVATest (f\_classif)

continuous |continuous | F-value (f\_regression)

It supports five selection modes: numTopFeatures, percentile, fpr, fdr, fwe:

* numTopFeatures chooses a fixed number of top features.
* percentile is similar to numTopFeatures but chooses a fraction of all features instead of a fixed number.
* fpr chooses all features whose p-values are below a threshold, thus controlling the false positive rate of selection.
* fdr uses the Benjamini-Hochberg procedure to choose all features whose false discovery rate is below a threshold.
* fwe chooses all features whose p-values are below a threshold. The threshold is scaled by 1/numFeatures, thus controlling the family-wise error rate of selection.

By default, the selection mode is numTopFeatures, with the default selectionThreshold sets to 50.

**Examples**

Assume that we have a DataFrame with the columns id, features, and label, which is used as our target to be predicted:

id | features | label

---|--------------------------------|---------

1 | [1.7, 4.4, 7.6, 5.8, 9.6, 2.3] | 3.0

2 | [8.8, 7.3, 5.7, 7.3, 2.2, 4.1] | 2.0

3 | [1.2, 9.5, 2.5, 3.1, 8.7, 2.5] | 3.0

4 | [3.7, 9.2, 6.1, 4.1, 7.5, 3.8] | 2.0

5 | [8.9, 5.2, 7.8, 8.3, 5.2, 3.0] | 4.0

6 | [7.9, 8.5, 9.2, 4.0, 9.4, 2.1] | 4.0

If we set featureType to continuous and labelType to categorical with numTopFeatures = 1, the last column in our features is chosen as the most useful feature:

id | features | label | selectedFeatures

---|--------------------------------|---------|------------------

1 | [1.7, 4.4, 7.6, 5.8, 9.6, 2.3] | 3.0 | [2.3]

2 | [8.8, 7.3, 5.7, 7.3, 2.2, 4.1] | 2.0 | [4.1]

3 | [1.2, 9.5, 2.5, 3.1, 8.7, 2.5] | 3.0 | [2.5]

4 | [3.7, 9.2, 6.1, 4.1, 7.5, 3.8] | 2.0 | [3.8]

5 | [8.9, 5.2, 7.8, 8.3, 5.2, 3.0] | 4.0 | [3.0]

6 | [7.9, 8.5, 9.2, 4.0, 9.4, 2.1] | 4.0 | [2.1]

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.UnivariateFeatureSelector**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Seq**(

(1, Vectors.dense(1.7, 4.4, 7.6, 5.8, 9.6, 2.3), 3.0),

(2, Vectors.dense(8.8, 7.3, 5.7, 7.3, 2.2, 4.1), 2.0),

(3, Vectors.dense(1.2, 9.5, 2.5, 3.1, 8.7, 2.5), 3.0),

(4, Vectors.dense(3.7, 9.2, 6.1, 4.1, 7.5, 3.8), 2.0),

(5, Vectors.dense(8.9, 5.2, 7.8, 8.3, 5.2, 3.0), 4.0),

(6, Vectors.dense(7.9, 8.5, 9.2, 4.0, 9.4, 2.1), 4.0)

)

**val** df **=** spark.createDataset(data).toDF("id", "features", "label")

**val** selector **=** **new** **UnivariateFeatureSelector**()

.setFeatureType("continuous")

.setLabelType("categorical")

.setSelectionMode("numTopFeatures")

.setSelectionThreshold(1)

.setFeaturesCol("features")

.setLabelCol("label")

.setOutputCol("selectedFeatures")

**val** result **=** selector.fit(df).transform(df)

println(s"UnivariateFeatureSelector output with top ${selector.getSelectionThreshold}" +

s" features selected using f\_classif")

result.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/UnivariateFeatureSelectorExample.scala" in the Spark repo.

**VarianceThresholdSelector**

VarianceThresholdSelector is a selector that removes low-variance features. Features with a variance not greater than the varianceThreshold will be removed. If not set, varianceThreshold defaults to 0, which means only features with variance 0 (i.e. features that have the same value in all samples) will be removed.

**Examples**

Assume that we have a DataFrame with the columns id and features, which is used as our target to be predicted:

id | features

---|--------------------------------

1 | [6.0, 7.0, 0.0, 7.0, 6.0, 0.0]

2 | [0.0, 9.0, 6.0, 0.0, 5.0, 9.0]

3 | [0.0, 9.0, 3.0, 0.0, 5.0, 5.0]

4 | [0.0, 9.0, 8.0, 5.0, 6.0, 4.0]

5 | [8.0, 9.0, 6.0, 5.0, 4.0, 4.0]

6 | [8.0, 9.0, 6.0, 0.0, 0.0, 0.0]

The variance for the 6 features are 16.67, 0.67, 8.17, 10.17, 5.07, and 11.47 respectively. If we use VarianceThresholdSelector with varianceThreshold = 8.0, then the features with variance <= 8.0 are removed:

id | features | selectedFeatures

---|--------------------------------|-------------------

1 | [6.0, 7.0, 0.0, 7.0, 6.0, 0.0] | [6.0,0.0,7.0,0.0]

2 | [0.0, 9.0, 6.0, 0.0, 5.0, 9.0] | [0.0,6.0,0.0,9.0]

3 | [0.0, 9.0, 3.0, 0.0, 5.0, 5.0] | [0.0,3.0,0.0,5.0]

4 | [0.0, 9.0, 8.0, 5.0, 6.0, 4.0] | [0.0,8.0,5.0,4.0]

5 | [8.0, 9.0, 6.0, 5.0, 4.0, 4.0] | [8.0,6.0,5.0,4.0]

6 | [8.0, 9.0, 6.0, 0.0, 0.0, 0.0] | [8.0,6.0,0.0,0.0]

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.VarianceThresholdSelector**

**import** **org.apache.spark.ml.linalg.Vectors**

**val** data **=** **Seq**(

(1, Vectors.dense(6.0, 7.0, 0.0, 7.0, 6.0, 0.0)),

(2, Vectors.dense(0.0, 9.0, 6.0, 0.0, 5.0, 9.0)),

(3, Vectors.dense(0.0, 9.0, 3.0, 0.0, 5.0, 5.0)),

(4, Vectors.dense(0.0, 9.0, 8.0, 5.0, 6.0, 4.0)),

(5, Vectors.dense(8.0, 9.0, 6.0, 5.0, 4.0, 4.0)),

(6, Vectors.dense(8.0, 9.0, 6.0, 0.0, 0.0, 0.0))

)

**val** df **=** spark.createDataset(data).toDF("id", "features")

**val** selector **=** **new** **VarianceThresholdSelector**()

.setVarianceThreshold(8.0)

.setFeaturesCol("features")

.setOutputCol("selectedFeatures")

**val** result **=** selector.fit(df).transform(df)

println(s"Output: Features with variance lower than" +

s" ${selector.getVarianceThreshold} are removed.")

result.show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/VarianceThresholdSelectorExample.scala" in the Spark repo.

**Locality Sensitive Hashing**

Locality Sensitive Hashing (LSH) is an important class of hashing techniques, which is commonly used in clustering, approximate nearest neighbor search and outlier detection with large datasets.

The general idea of LSH is to use a family of functions (“LSH families”) to hash data points into buckets, so that the data points which are close to each other are in the same buckets with high probability, while data points that are far away from each other are very likely in different buckets. An LSH family is formally defined as follows.

In a metric space (M, d), where M is a set and d is a distance function on M, an LSH family is a family of functions h that satisfy the following properties:

∀p,q∈M,d(p,q)≤r1⇒Pr(h(p)=h(q))≥p1d(p,q)≥r2⇒Pr(h(p)=h(q))≤p2∀p,q∈M,d(p,q)≤r1⇒Pr(h(p)=h(q))≥p1d(p,q)≥r2⇒Pr(h(p)=h(q))≤p2

This LSH family is called (r1, r2, p1, p2)-sensitive.

In Spark, different LSH families are implemented in separate classes (e.g., MinHash), and APIs for feature transformation, approximate similarity join and approximate nearest neighbor are provided in each class.

In LSH, we define a false positive as a pair of distant input features (with d(p,q)≥r2d(p,q)≥r2) which are hashed into the same bucket, and we define a false negative as a pair of nearby features (with d(p,q)≤r1d(p,q)≤r1) which are hashed into different buckets.

**LSH Operations**

We describe the major types of operations which LSH can be used for. A fitted LSH model has methods for each of these operations.

**Feature Transformation**

Feature transformation is the basic functionality to add hashed values as a new column. This can be useful for dimensionality reduction. Users can specify input and output column names by setting inputCol and outputCol.

LSH also supports multiple LSH hash tables. Users can specify the number of hash tables by setting numHashTables. This is also used for OR-amplification in approximate similarity join and approximate nearest neighbor. Increasing the number of hash tables will increase the accuracy but will also increase communication cost and running time.

The type of outputCol is Seq[Vector] where the dimension of the array equals numHashTables, and the dimensions of the vectors are currently set to 1. In future releases, we will implement AND-amplification so that users can specify the dimensions of these vectors.

**Approximate Similarity Join**

Approximate similarity join takes two datasets and approximately returns pairs of rows in the datasets whose distance is smaller than a user-defined threshold. Approximate similarity join supports both joining two different datasets and self-joining. Self-joining will produce some duplicate pairs.

Approximate similarity join accepts both transformed and untransformed datasets as input. If an untransformed dataset is used, it will be transformed automatically. In this case, the hash signature will be created as outputCol.

In the joined dataset, the origin datasets can be queried in datasetA and datasetB. A distance column will be added to the output dataset to show the true distance between each pair of rows returned.

**Approximate Nearest Neighbor Search**

Approximate nearest neighbor search takes a dataset (of feature vectors) and a key (a single feature vector), and it approximately returns a specified number of rows in the dataset that are closest to the vector.

Approximate nearest neighbor search accepts both transformed and untransformed datasets as input. If an untransformed dataset is used, it will be transformed automatically. In this case, the hash signature will be created as outputCol.

A distance column will be added to the output dataset to show the true distance between each output row and the searched key.

**Note:** Approximate nearest neighbor search will return fewer than k rows when there are not enough candidates in the hash bucket.

**LSH Algorithms**

**Bucketed Random Projection for Euclidean Distance**

Bucketed Random Projection is an LSH family for Euclidean distance. The Euclidean distance is defined as follows:

d(x,y)=∑i(xi−yi)2−−−−−−−−−−√d(x,y)=∑i(xi−yi)2

Its LSH family projects feature vectors xx onto a random unit vector vv and portions the projected results into hash buckets:

h(x)=⌊x⋅vr⌋h(x)=⌊x⋅vr⌋

where r is a user-defined bucket length. The bucket length can be used to control the average size of hash buckets (and thus the number of buckets). A larger bucket length (i.e., fewer buckets) increases the probability of features being hashed to the same bucket (increasing the numbers of true and false positives).

Bucketed Random Projection accepts arbitrary vectors as input features, and supports both sparse and dense vectors.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.BucketedRandomProjectionLSH**

**import** **org.apache.spark.ml.linalg.Vectors**

**import** **org.apache.spark.sql.SparkSession**

**import** **org.apache.spark.sql.functions.col**

**val** dfA **=** spark.createDataFrame(**Seq**(

(0, Vectors.dense(1.0, 1.0)),

(1, Vectors.dense(1.0, -1.0)),

(2, Vectors.dense(-1.0, -1.0)),

(3, Vectors.dense(-1.0, 1.0))

)).toDF("id", "features")

**val** dfB **=** spark.createDataFrame(**Seq**(

(4, Vectors.dense(1.0, 0.0)),

(5, Vectors.dense(-1.0, 0.0)),

(6, Vectors.dense(0.0, 1.0)),

(7, Vectors.dense(0.0, -1.0))

)).toDF("id", "features")

**val** key **=** Vectors.dense(1.0, 0.0)

**val** brp **=** **new** **BucketedRandomProjectionLSH**()

.setBucketLength(2.0)

.setNumHashTables(3)

.setInputCol("features")

.setOutputCol("hashes")

**val** model **=** brp.fit(dfA)

*// Feature Transformation*

println("The hashed dataset where hashed values are stored in the column 'hashes':")

model.transform(dfA).show()

*// Compute the locality sensitive hashes for the input rows, then perform approximate*

*// similarity join.*

*// We could avoid computing hashes by passing in the already-transformed dataset, e.g.*

*// `model.approxSimilarityJoin(transformedA, transformedB, 1.5)`*

println("Approximately joining dfA and dfB on Euclidean distance smaller than 1.5:")

model.approxSimilarityJoin(dfA, dfB, 1.5, "EuclideanDistance")

.select(col("datasetA.id").alias("idA"),

col("datasetB.id").alias("idB"),

col("EuclideanDistance")).show()

*// Compute the locality sensitive hashes for the input rows, then perform approximate nearest*

*// neighbor search.*

*// We could avoid computing hashes by passing in the already-transformed dataset, e.g.*

*// `model.approxNearestNeighbors(transformedA, key, 2)`*

println("Approximately searching dfA for 2 nearest neighbors of the key:")

model.approxNearestNeighbors(dfA, key, 2).show()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/BucketedRandomProjectionLSHExample.scala" in the Spark repo.

**MinHash for Jaccard Distance**

MinHash is an LSH family for Jaccard distance where input features are sets of natural numbers. Jaccard distance of two sets is defined by the cardinality of their intersection and union:

d(A,B)=1−|A∩B||A∪B|d(A,B)=1−|A∩B||A∪B|

MinHash applies a random hash function g to each element in the set and take the minimum of all hashed values:

h(A)=mina∈A(g(a))h(A)=mina∈A(g(a))

The input sets for MinHash are represented as binary vectors, where the vector indices represent the elements themselves and the non-zero values in the vector represent the presence of that element in the set. While both dense and sparse vectors are supported, typically sparse vectors are recommended for efficiency. For example, Vectors.sparse(10, Array[(2, 1.0), (3, 1.0), (5, 1.0)]) means there are 10 elements in the space. This set contains elem 2, elem 3 and elem 5. All non-zero values are treated as binary “1” values.

**Note:** Empty sets cannot be transformed by MinHash, which means any input vector must have at least 1 non-zero entry.

* **Scala**
* **Java**
* **Python**

**import** **org.apache.spark.ml.feature.MinHashLSH**

**import** **org.apache.spark.ml.linalg.Vectors**

**import** **org.apache.spark.sql.SparkSession**

**import** **org.apache.spark.sql.functions.col**

**val** dfA **=** spark.createDataFrame(**Seq**(

(0, Vectors.sparse(6, **Seq**((0, 1.0), (1, 1.0), (2, 1.0)))),

(1, Vectors.sparse(6, **Seq**((2, 1.0), (3, 1.0), (4, 1.0)))),

(2, Vectors.sparse(6, **Seq**((0, 1.0), (2, 1.0), (4, 1.0))))

)).toDF("id", "features")

**val** dfB **=** spark.createDataFrame(**Seq**(

(3, Vectors.sparse(6, **Seq**((1, 1.0), (3, 1.0), (5, 1.0)))),

(4, Vectors.sparse(6, **Seq**((2, 1.0), (3, 1.0), (5, 1.0)))),

(5, Vectors.sparse(6, **Seq**((1, 1.0), (2, 1.0), (4, 1.0))))

)).toDF("id", "features")

**val** key **=** Vectors.sparse(6, **Seq**((1, 1.0), (3, 1.0)))

**val** mh **=** **new** **MinHashLSH**()

.setNumHashTables(5)

.setInputCol("features")

.setOutputCol("hashes")

**val** model **=** mh.fit(dfA)

*// Feature Transformation*

println("The hashed dataset where hashed values are stored in the column 'hashes':")

model.transform(dfA).show()

*// Compute the locality sensitive hashes for the input rows, then perform approximate*

*// similarity join.*

*// We could avoid computing hashes by passing in the already-transformed dataset, e.g.*

*// `model.approxSimilarityJoin(transformedA, transformedB, 0.6)`*

println("Approximately joining dfA and dfB on Jaccard distance smaller than 0.6:")

model.approxSimilarityJoin(dfA, dfB, 0.6, "JaccardDistance")

.select(col("datasetA.id").alias("idA"),

col("datasetB.id").alias("idB"),

col("JaccardDistance")).show()

*// Compute the locality sensitive hashes for the input rows, then perform approximate nearest*

*// neighbor search.*

*// We could avoid computing hashes by passing in the already-transformed dataset, e.g.*

*// `model.approxNearestNeighbors(transformedA, key, 2)`*

*// It may return less than 2 rows when not enough approximate near-neighbor candidates are*

*// found.*

println("Approximately searching dfA for 2 nearest neighbors of the key:")

model.approxNearestNeighbors(dfA, key, 2).show()

================================================================================

* **estimator:** estimator is the component which delivers a model, fitting an algorithm to train data. Indeed *fit()* is the key method for an estimator and produces, as said a *model* which is a transformer. Leveraging the parallel processing which is provided by Spark, it is possible to run several estimators in parallel on different training dataset in order to find the best solution (or even to avoid overfitting issue). *ML algorithms* are basically a set of Estimators, they build a rich set of machine learning (ML) common algorithms, available from MLlib. This is a library of algorithms meant to be scalable and run in a parallel environment. Starting from the 2.0 release of Spark, the RDD-based library is in maintenance mode (the RDD-based APIs are expected to be removed in 3.0 release) whereas the mainstream development is focused on supporting dataframes. In MLlib features are to be expressed with *labeledpoints*, which means numeric vectors for features and predictors.Pipelines of transformers are, even for this reason, extremely useful to operationalize an ML solutions Spark-based. For additional details on MLlib refer to Apache Spark documentation

In this post we’ll see a simple example of pipeline usage and we’ll see two version of the same example: the first one using Scala (which is a kind of “native” language for Spark environment), afterward we’ll see how to implement the same example in R, leveraging SaprklyR package in order to show how powerful and complete it is.

**The dataset**

For this example, the dataset comes from UCI – Machine Learning Repository Irvine, CA: University of California, School of Information and Computer Science. “Adults Income” dataset reports individual’s annual income results from various factors. The annual income will be our label (it is divided into two classes: <=50K and >50K) and there are 14 features, for each individual, we can leverage to explore the possibility in predicting income level. For additional detail on “adults dataset” refer to the UCI machine learning repository http://www.cs.toronto.edu/~delve/data/adult/desc.html.

**Scala code**

As said, we’ll show how we can use scala API to access pipeline in MLlib, therefore we need to include references to classes we’re planning to use in our example and to start a spark session :

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.functions.\_

import org.apache.spark.sql.\_

import org.apache.spark.ml.Pipeline

import org.apache.spark.ml.feature.{VectorAssembler, StringIndexer, VectorIndexer}

import org.apache.spark.ml.classification.LogisticRegression

import org.apache.spark.sql.SparkSession

then we’ll read dataset and will start to manipulate data in order to prepare for the pipeline. In our example, we’ll get data out of local repository (instead of referring to an eg. HDFS or Datalake repository, there are API – for both scala and R – which allows the access to these repositories as well). We’ll leverage this upload activity also to rename some columns, in particular, we’ll rename the “income” column as “label” since we’ll use this a label column in our logistic regression algorithm.

//load data source from local repository

val csv = spark.read.option("inferSchema","true")

.option("header", "true").csv("...\yyyy\xxxx\adult.csv")

val data\_start = csv.select(($"workclass"),($"gender"),($"education"),($"age"),

($"marital-status").alias("marital"), ($"occupation"),($"relationship"),

($"race"),($"hours-per-week").alias("hr\_per\_week"), ($"native-country").alias("country"),

($"income").alias("label"),($"capital-gain").alias("capitalgain"),

($"capital-loss").alias("capitalloss"),($"educational-num").alias("eduyears")).toDF

We’ll do some data clean up basically recoding the “working class” and “marital” columns, in order to reduce the number of codes and we’ll get rid of rows for which “occupation”, “working class”” (even recoded) and “capital gain” are not available. For first two column the dataset has the “?” value instead of “NA”, for capital gain there’s the 99999 value which will be filtered out. To recode “working class” and “marital” columns we’ll use UDF functions which in turn are wrappers of the actual recoding functions. To add a new column to the (new) dataframe we’ll use the “withColumn” method which will add “new\_marital” and “new\_workclass” to the *startingdata* dataframe. Afterwards, we’ll filter out all missing values and we’ll be ready to build the pipeline.

// recoding marital status and working class, adding a new column

def newcol\_marital(str1:String): String = {

var nw\_str = "noVal"

if ((str1 == "Married-spouse-absent") | (str1 =="Married-AF-spouse")

| (str1 == "Married-civ-spouse")) {nw\_str = "Married" }

else if ((str1 == "Divorced") | (str1 == "Never-married" )

| (str1 == "Separated" ) | (str1 == "Widowed"))

{nw\_str = "Nonmarried" }

else { nw\_str = str1}

return nw\_str

}

val udfnewcol = udf(newcol\_marital \_)

val recodeddata = data\_start.withColumn("new\_marital", udfnewcol('marital'))

def newcol\_wkclass(str1:String): String = {

var nw\_str = "noVal"

if ((str1 == "Local-gov") | (str1 =="Federal-gov") | (str1 == "State-gov"))

{nw\_str = "Gov" }

else if ((str1 == "Self-emp-not-inc") | (str1 == "Self-emp-inc" ))

{nw\_str = "Selfemployed" }

else if ((str1 == "Never-worked") | (str1 == "Without-pay"))

{nw\_str = "Notemployed" }

else { nw\_str = str1}

return nw\_str

}

val udfnewcol = udf(newcol\_wkclass \_)

val startingdata = recodeddata.withColumn("new\_workclass", udfnewcol('workclass'))

// remove missing data

val df\_work01 = startingdata.na.drop("any")

val df\_work = startingdata.filter("occupation <> '?'

and capitalgain < 99999

and new\_workclass <> '?'

and country <> '?' ")

In our example, we’re going to use 12 features, 7 are categorical variables and 5 are numeric variables. The feature’s array we’ll use to fit the model will be the results of merging two arrays, one for categorical variables and the second one for numeric variables. Before building the categorical variables array, we need to transform categories to indexes using transformers provided by spark.ml, even the *label* must be transformed to an index. Our pipeline then will include 7 pipeline *stages* to transform categorical variables, 1 *stage* to transform the label, 2 *stages* to build categorical and numeric arrays, and the final *stage* to fit the logistic model. Finally, we’ll build an 11-stages pipeline.  
To transform categorical variables into index we’ll use “Stringindexer” *Transformer*. StringIndexer encodes a vector of string to a column of non-negative indices, ranging from 0 to the number of values. The indices ordered by label frequencies, so the most frequent value gets index 0. For each variable, we need to define the *input column* and the *output column* which we’ll use as input for other transformer or evaluators. Finally it is possible to define the strategy to handle unseen labels (possible when you use the S*tringindexer* to fit a model and run the fitted model against a test dataframe) through *setHandleInvalid* method , in our case we simply put “skip” to tell *Stringindexer* we want to skip unseen labels (additional details are available in MLlib documentation).

// define stages

val new\_workclassIdx = new StringIndexer().setInputCol("new\_workclass")

.setOutputCol("new\_workclassIdx").setHandleInvalid("skip")

val genderIdx = new StringIndexer().setInputCol("gender")

.setOutputCol("genderIdx").setHandleInvalid("skip")

val maritalIdx = new StringIndexer().setInputCol("new\_marital")

.setOutputCol("maritalIdx").setHandleInvalid("skip")

val occupationIdx = new StringIndexer().setInputCol("occupation")

.setOutputCol("occupationIdx").setHandleInvalid("skip")

val relationshipIdx = new StringIndexer().setInputCol("relationship")

.setOutputCol("relationshipIdx").setHandleInvalid("skip")

val raceIdx = new StringIndexer().setInputCol("race")

.setOutputCol("raceIdx").setHandleInvalid("skip")

val countryIdx = new StringIndexer().setInputCol("country")

.setOutputCol("countryIdx").setHandleInvalid("skip")

val labelIdx = new StringIndexer().setInputCol("label")

.setOutputCol("labelIdx").setHandleInvalid("skip")

In addition to *Transfomer* and *Estimator* provided by spark.ml package, it is possible to define custom Estimator and Transformers. As an example we’ll see how to define a custom transformer aimed at recoding “marital status” in our dataset (basically we’ll do the same task we have already seen, implementing it with a custom transformer; for additional details on implementing customer estimator and transformer. To define a custom transformer, we’ll define a new scala class, *columnRecoder*, which extends the Transformer class, we’ll override the *transformSchema*method to map the correct type of the new column we’re going to add with the transformation and we’ll implement the *transform* method which actually does the recoding in our dataset. A possible implementation is :

import org.apache.spark.ml.Transformer

class columnRecoder(override val uid: String) extends Transformer {

final val inputCol= new Param[String](this, "inputCol", "input column")

final val outputCol = new Param[String](this, "outputCol", "output column")

def setInputCol(value: String): this.type = set(inputCol, value)

def setOutputCol(value: String): this.type = set(outputCol, value)

def this() = this(Identifiable.randomUID("columnRecoder"))

def copy(existingParam: ParamMap): columnRecoder = {defaultCopy(existingParam)}

override def transformSchema(schema: StructType): StructType = {

// Check inputCol type

val idx = schema.fieldIndex($(inputCol))

val field = schema.fields(idx)

if (field.dataType != StringType) {

throw new Exception(s"Input type ${field.dataType} type mismatch: String expected")

}

// The return field

schema.add(StructField($(outputCol),StringType, false))

}

val newcol\_recode = new marital\_code()

private def newcol\_recode(str1: String): String = {

var nw\_str = "noVal"

if ((str1 == "Married-spouse-absent") | (str1 =="Married-AF-spouse")

| (str1 == "Married-civ-spouse"))

{nw\_str = "Married" }

else if ((str1 == "Divorced") | (str1 == "Never-married" )

| (str1 == "Separated" ) | (str1 == "Widowed"))

{nw\_str = "Nonmarried" }

else { nw\_str = str1}

nw\_str

}

private def udfnewcol = udf(newcol\_recode.recode(\_))

def transform(df: Dataset[\_]): DataFrame = {

df.withColumn($(outputCol), udfnewcol(df($(inputCol))))

}

}

Once defined as a transformer, we can use it in our pipeline as the first stage.

// define stages

val new\_marital = new columnRecoder().setInputCol("marital")

.setOutputCol("new\_marital")

val new\_workclassIdx = new StringIndexer().setInputCol("new\_workclass")

.setOutputCol("new\_workclassIdx").setHandleInvalid("skip")

val genderIdx = new StringIndexer().setInputCol("gender")

.setOutputCol("genderIdx").setHandleInvalid("skip")

val maritalIdx = new StringIndexer().setInputCol("new\_marital")

.setOutputCol("maritalIdx").setHandleInvalid("skip")

.......

A second step in building our pipeline is to assemble categorical indexes in a single vector, therefore many categorical indexes are put all together in a single vector through the *VectorAssembler*transformer. This *VectorAssembler* will deliver a single column feature which will be, in turn, transformed to indexes by *VectorIndexer* transformer to deliver indexes within the *“catFeatures”* column:

// cat vector for categorical variables

val catVect = new VectorAssembler()

.setInputCols(Array("new\_workclassIdx", "genderIdx", "catVect","maritalIdx", "occupationIdx","relationshipIdx","raceIdx","countryIdx"))

.setOutputCol("cat01Features")

val catIdx = new VectorIndexer()

.setInputCol(catVect.getOutputCol)

.setOutputCol("catFeatures")

For numeric variables we need just to assemble columns with VectorAssembler, then we’re ready to put these two vectors (one for categorical variables, the other for numeric variables) together in a single vector.

// numeric vector for numeric variable

val numVect = new VectorAssembler()

.setInputCols(Array("age","hr\_per\_week","capitalgain","capitalloss","eduyears"))

.setOutputCol("numFeatures")

val featVect = new VectorAssembler()

.setInputCols(Array("catFeatures", "numFeatures"))

.setOutputCol("features")

We have now label and features ready to build the logistic regression model which is the final component of our pipeline. We can also set some parameters for the model, in particular, we can define the threshold (by default set to 0.5) to make the decision between label values, as well as the max number of iterations for this algorithm and a parameter to tune regularization.  
When all stages of the pipeline are ready, we just need to define the *pipeline* component itself, passing as an input parameter an array with all defined stages:

val lr = new LogisticRegression().setLabelCol("labelIdx").setFeaturesCol("features")

.setThreshold(0.33).setMaxIter(10).setRegParam(0.2)

val pipeline = new Pipeline().setStages(Array(new\_marital,new\_workclassIdx, labelIdx,maritalIdx,occupationIdx, relationshipIdx,raceIdx,genderIdx, countryIdx,catVect, catIdx, numVect,featVect,lr))

Now the pipeline component, which encompasses a number of transformations as well as the classification algorithm, is ready; to actually use it we supply a *train* dataset to fit the model and then a *test* dataset to evaluate our fitted model. Since we have defined a pipeline, we’ll be sure that both, train and test datasets, will undergo the same transformations, therefore, we don’t have to replicate the process twice.  
We need now to define train and test datasets.In our dataset, label values are unbalanced being the “more than 50k USD per year” value around the 25% of the total, in order to preserve the same proportion between label values we’ll subset the original dataset based on label value, obtaining a *low-income* dataset and an *high-income* dataset. We’ll split both dataset for train (70%) and test (30%), then we’ll merge back the two “train”” and the two “test” datasets and we’ll use resulting “train” dataset as input for our pipeline:

// split betwen train and test

val df\_low\_income = df\_work.filter("label == '<=50K'")

val df\_high\_income = df\_work.filter("label == '>50K'")

val splits\_LI = df\_low\_income.randomSplit(Array(0.7, 0.3), seed=123)

val splits\_HI = df\_high\_income.randomSplit(Array(0.7, 0.3), seed=123)

val df\_work\_train = splits\_LI(0).unionAll(splits\_HI(0))

val df\_work\_test = splits\_LI(1).unionAll(splits\_HI(1))

// fitting the pipeline

val data\_model = pipeline.fit(df\_work\_train)

Once the pipeline is trained, we can use the data\_model for testing against the test dataset, calculate the confusion matrix and evaluate the classifier metrics :

// generate prediction

val data\_prediction = data\_model.transform(df\_work\_test)

val data\_predicted = data\_prediction.select("features", "prediction", "label","labelIdx")

// confusion matrix

val tp = data\_predicted.filter("prediction == 1 AND labelIdx == 1").count().toFloat

val fp = data\_predicted.filter("prediction == 1 AND labelIdx == 0").count().toFloat

val tn = data\_predicted.filter("prediction == 0 AND labelIdx == 0").count().toFloat

val fn = data\_predicted.filter("prediction == 0 AND labelIdx == 1").count().toFloat

val metrics = spark.createDataFrame(Seq(

("TP", tp),

("FP", fp),

("TN", tn),

("FN", fn),

("Precision", tp / (tp + fp)),

("Recall", tp / (tp + fn)))).toDF("metric", "value")

metrics.show()

**R code and SparklyR**

Now we’ll try to replicate the same example we just saw in R, more precisely, working with the SparklyR package.  
We’ll use the developer version of SparklyR (as you possibly know, there’s an interesting debate on the best API to access Apache Spark resources from R. We need to install it from github before connecting with Spark environment.  
In our case Spark is a standalone instance running version 2.2.0, as reported in the official documentation for SparklyR, configuration parameters can be set through spark\_config() function, in particular, spark\_config() provides the basic configuration used by default for spark connection. To change parameters it’s possible to get default configuration via spark\_connection() then change parameters as needed

library(sparklyr)

library(dplyr)

sc <- spark\_connect(master = "local",spark\_home = "...\Local\spark",version="2.2.0")

After connecting with Spark, we’ll read the dataset into a Spark dataframe and select fields (with column renaming, where needed) we’re going to use for our classification example. It is worthwhile to remember that *dplyr* (and therefore sparklyR) uses lazy evaluation when accessing and manipulating data, which means that *‘the data is only fetched at the last moment when it’s needed’* (Efficient R programming, *C.Gillespie R.Lovelace*).For this reason, later on we’ll that we’ll force the statement execution calling *action* functions (like *collect()*).  
As we did in scala script, we’ll recode *“marital status”* and *“workclass”* in order to simplify the analysis. In renaming dataframe columns, we’ll use the “select” function available from dplyr package, indeed one of the SparklyR aims is to allow the manipulation of Spark dataframes/tables through dplyr functions. This is an example of how function like “select” or “filter” can be used also for spark dataframes.

income\_table <- spark\_read\_csv(sc,"income\_table","...\adultincome\adult.csv")

income\_table <- select(income\_table,"workclass","gender","eduyears"="educationalnum",

"age","marital"="maritalstatus","occupation","relationship",

"race","hr\_per\_week"="hoursperweek","country"="nativecountry",

"label"="income","capitalgain","capitalloss")

# recoding marital status and workingclass

income\_table <- income\_table %>% mutate(marital = ifelse(marital == "Married-spouse-absent" | marital == "Married-AF-spouse" |

marital == "Married-civ-spouse","married","nonMarried"))

income\_table <- income\_table %>% mutate(workclass = ifelse(workclass == "Local-gov"| workclass == "Federal-gov" | workclass == "State\_gov",

"Gov",ifelse(workclass == "Self-emp-inc" | workclass == "Self-emp-not-inc","Selfemployed",ifelse(workclass=="Never-worked" |

workclass == "Without-pay","Notemployed",workclass))))

SparklyR provides functions which are bound to Spark spark.ml package, therefore it is possible to build Machine Learning solutions putting together the power of dplyr grammar with Spark ML algorithms. To simply link package function to Spark.ml, SparklyR provides functions which use specific prefixes to identify functions group:

* functions prefixed with **sdf\_** generally access the Scala Spark DataFrame API directly (as opposed to the dplyr interface which uses Spark SQL) to manipulate dataframes;
* functions prefixed with **ft\_** are functions to manipulate and transform features. Pipeline transformers and estimators belong to this group of functions;
* functions prefixed with **ml\_** implement algorithms to build machine learning workflow. Even pipeline instance is provided by *ml\_pipeline()* which belongs to these functions.

We can then proceed with pipeline, stages and feature array definition. Ft-prefixed functions recall the spark.ml functions these are bound to:

income\_pipe <- ml\_pipeline(sc,uid="income\_pipe")

income\_pipe <-ft\_string\_indexer(income\_pipe,input\_col="workclass",output\_col="workclassIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="gender",output\_col="genderIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="marital",output\_col="maritalIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="occupation",output\_col="occupationIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="race",output\_col="raceIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="country",output\_col="countryIdx")

income\_pipe <- ft\_string\_indexer(income\_pipe,input\_col="label",output\_col="labelIdx")

array\_features <- c("workclassIdx","genderIdx","maritalIdx",

"occupationIdx","raceIdx","countryIdx","eduyears",

"age","hr\_per\_week","capitalgain","capitalloss")

income\_pipe <- ft\_vector\_assembler(income\_pipe, input\_col=array\_features, output\_col="features")

In our example we’ll use *ml\_logistic\_regression()* to implement the classification solution aimed at predicting the income level. Being bound to the *LogisticRegression()* function in spark.ml package, (as expected) all parameters are available for specific setting, therefore we can can “mirror” the same call we made in scala code: e.g. we can manage the “decision” threshold for our binary classifier (setting the value to 0.33).

# putting in pipe the logistic regression evaluator

income\_pipe <- ml\_logistic\_regression(income\_pipe,

features\_col = "features", label\_col = "labelIdx",

family= "binomial",threshold = 0.33, reg\_param=0.2, max\_iter=10L)

Final steps in our example are to split data between train and test subset, fit the pipeline and evaluate the classifier. As we’ve had already done in scala code, we’ll manage the unbalance in label values, splitting the dataframe in a way which secures the relative percentage of label values. Afterwards, we’ll fit the pipeline and get the predictions relative to test dataframe (as we did already in scala code).

# data split

# dealing with label inbalance

df\_low\_income = filter(income\_table,income\_table$label == "<=50K")

df\_high\_income = filter(income\_table,income\_table$label == ">50K")

splits\_LI <- sdf\_partition(df\_low\_income, test=0.3, train=0.7, seed = 7711)

splits\_HI <- sdf\_partition(df\_high\_income,test=0.3,train=0.7,seed=7711)

df\_test <- sdf\_bind\_rows(splits\_LI[[1]],splits\_HI[[1]])

df\_train <- sdf\_bind\_rows(splits\_LI[[2]],splits\_HI[[2]])

df\_model <- ml\_fit(income\_pipe,df\_train)

Once fitted, the model exposes, for each pipeline stage, all parameters and logistic regression (which is the last element in the stages list) coefficients.

df\_model$stages

df\_model$stages[[9]]$coefficients

We can then process the test dataset putting it in the pipeline to get predictions and evaluate the fitted model

df\_prediction <- ml\_predict(df\_model,df\_test)

df\_results <- select(df\_prediction,"prediction","labelIdx","probability")

We can then proceed to evaluate the confusion matrix and get main metrics to evaluate the model (from precision to AUC).

# calculating confusion matrix

df\_tp <- filter(df\_results,(prediction == 1 && labelIdx == 1))

df\_fp <- filter(df\_results,(prediction ==1 && labelIdx == 0))

df\_tn <- filter(df\_results,(prediction == 1 && labelIdx == 0))

df\_fn <- filter(df\_results,(prediction == 1 && labelIdx == 1))

tp <- df\_tp %>% tally() %>% collect() %>% as.integer()

fp <- df\_fp %>% tally() %>% collect() %>% as.integer()

tn <- df\_tn %>% tally() %>% collect() %>% as.integer()

fn <- df\_fn %>% tally() %>% collect() %>% as.integer()

df\_precision <- (tp/(tp+fp))

df\_recall <- (tp/(tp+fn))

df\_accuracy = (tp+tn)/(tp+tn+fp+fn)

c\_AUC <- ml\_binary\_classification\_eval(df\_prediction, label\_col = "labelIdx",

prediction\_col = "prediction", metric\_name = "areaUnderROC")

**Conclusion**

As we have seen, pipelines are a useful mechanism to assemble and serialize transformation in order to make it repeatable for different sets of data. Are then a simple way for fitting and evaluating models through train/test datasets, but also suitable to run the same sequence of transformer/estimator in parallel over different nodes of the Spark cluster (i.e. to find the best parameter set). Moreover, a powerful API to deal with pipelines in R is available by SparklyR package, which provides in addition a comprehensive set of functions to leverage the ML spark package. Finally a support to run R code distributed across a Spark cluster has been to SparklyR with the *spark\_apply()* function which makes evenmore interesting the possibility to leverage pipelines in R in ditributed environment for analytical solutions.