Extracting, transforming and selecting features - spark.ml

This section covers algorithms for working with features, roughly divided into these groups:

- Extraction: Extracting features from “raw” data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features

Table of Contents

Feature Extractors

TF-IDF (HashingTF and IDF)

Term Frequency-Inverse Document Frequency (TF-IDF) is a common text pre-processing step. In Spark ML, TF-IDF is separate into two parts: TF (+hashing) and IDF.

TF: `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. The algorithm combines Term Frequency (TF) counts with the hashing trick for dimensionality reduction.

IDF: `IDF` is an `Estimator` which fits on a dataset and produces an `IDFModel`. The `IDFModel` takes feature vectors (generally created from `HashingTF`) and scales each column. Intuitively, it down-weights columns which appear frequently in a corpus.

Please refer to the MLlib user guide on TF-IDF for more details on Term Frequency and Inverse Document Frequency.

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**

Refer to the HashingTF Scala docs and the IDF Scala docs for more details on the API.

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

val sentenceData = sqlContext.createDataFrame(Seq(
  (0, "Hi I heard about Spark"),
  (0, "I wish Java could use case classes"),
  (1, "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)
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)
val rescaledData = idfModel.transform(featurizedData)
rescaledData.select("features", "label").take(3).foreach(println)
```

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 for as features for prediction, document similarity calculations, etc. Please refer to the MLlib user guide on Word2Vec for more details.

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**

Refer to the Word2Vec Scala docs for more details on the API.

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

// Input data: Each row is a bag of words from a sentence or document.
val documentDF = sqlContext.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.select("result").take(3).foreach(println)
```

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 affect 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.

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**

Refer to the CountVectorizer Scala docs and the CountVectorizerModel Scala docs for more details on the API.

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

val df = sqlContext.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).select("features").show()
```

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

Scala **Java** **Python**

Refer to the Tokenizer Scala docs and the RegexTokenizer Scala docs for more details on the API.

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

val sentenceDataFrame = sqlContext.createDataFrame(Seq(
  (0, "Hi I heard about Spark"),
  (1, "I wish Java could use case classes"),
  (2, "Logistic,regression,models,are,neat")
)).toDF("label", "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 tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("words", "label").take(3).foreach(println)
val regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("words", "label").take(3).foreach(println)
```

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. We provide a list of stop words by default, accessible by calling `getStopWords` on a newly instantiated `StopWordsRemover` instance. 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, baloon]
 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, baloon]  |  [saw, red, baloon]
 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**

Refer to the StopWordsRemover Scala docs for more details on the API.

```scala
import org.apache.spark.ml.feature.StopWordsRemover

val remover = new StopWordsRemover()
  .setInputCol("raw")
  .setOutputCol("filtered")

val dataSet = sqlContext.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "baloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb"))
)).toDF("id", "raw")

remover.transform(dataSet).show()
```

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

n-gram

An n-gram is a sequence of n tokens (typically words) for some integer n. The `NGram` class can be used to transform input features into n-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 n-gram. The output will consist of a sequence of n-grams where each n-gram is represented by a space-delimited string of n consecutive words. If the input sequence contains fewer than `n` strings, no output is produced.

Scala **Java** **Python**

Refer to the NGram Scala docs for more details on the API.

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

val wordDataFrame = sqlContext.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("label", "words")

val ngram = new NGram().setInputCol("words").setOutputCol("ngrams")
val ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.take(3).map(_.getAs[Stream[String]]("ngrams").toList).foreach(println)
```

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.

Scala **Java** **Python**

Refer to the Binarizer Scala docs for more details on the API.

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

val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataFrame: DataFrame = sqlContext.createDataFrame(data).toDF("label", "feature")

val binarizer: Binarizer = new Binarizer()
  .setInputCol("feature")
  .setOutputCol("binarized_feature")
  .setThreshold(0.5)

val binarizedDataFrame = binarizer.transform(dataFrame)
val binarizedFeatures = binarizedDataFrame.select("binarized_feature")
binarizedFeatures.collect().foreach(println)
```

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.

Scala **Java** **Python**

Refer to the PCA Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.PCA
import org.apache.spark.mllib.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 = sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")
val pca = new PCA()
  .setInputCol("features")
  .setOutputCol("pcaFeatures")
  .setK(3)
  .fit(df)
val pcaDF = pca.transform(df)
val result = pcaDF.select("pcaFeatures")
result.show()
```

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.

Scala **Java** **Python**

Refer to the PolynomialExpansion Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.PolynomialExpansion
import org.apache.spark.mllib.linalg.Vectors

val data = Array(
  Vectors.dense(-2.0, 2.3),
  Vectors.dense(0.0, 0.0),
  Vectors.dense(0.6, -1.1)
)
val df = sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")
val polynomialExpansion = new PolynomialExpansion()
  .setInputCol("features")
  .setOutputCol("polyFeatures")
  .setDegree(3)
val polyDF = polynomialExpansion.transform(df)
polyDF.select("polyFeatures").take(3).foreach(println)
```

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 N real-valued sequence in the time domain into another length N real-valued sequence in the frequency domain. A DCT class provides this functionality, implementing the DCT-II and scaling the result by $1/\sqrt{2}$ such that the representing matrix for the transform is unitary. No shift is applied to the transformed sequence (e.g. the 0th element of the transformed sequence is the 0th DCT coefficient and *not* the $N/2$th).

Scala **Java**

Refer to the DCT Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.DCT
import org.apache.spark.mllib.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 = sqlContext.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(3)
```

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. The indices are in `[0, numLabels)`, ordered by label frequencies. So the most frequent label gets index `0`. 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`.

Additionaly, there are two 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

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
```

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 row containing "d" does not appear.

Scala **Java** **Python**

Refer to the StringIndexer Scala docs for more details on the API.

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

val df = sqlContext.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. The 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

Refer to the IndexToString Scala docs for more details on the API.

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

val df = sqlContext.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)

val converter = new IndexToString()
  .setInputCol("categoryIndex")
  .setOutputCol("originalCategory")

val converted = converter.transform(indexed)
converted.select("id", "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 column of label indices to a column of binary vectors, with at most a single one-value. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features

Scala **Java** **Python**

Refer to the OneHotEncoder Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val df = sqlContext.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)

val encoder = new OneHotEncoder()
  .setInputCol("categoryIndex")
  .setOutputCol("categoryVec")
val encoded = encoder.transform(indexed)
encoded.select("id", "categoryVec").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 `Vector`s. 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.

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

Refer to the VectorIndexer Scala docs for more details on the API.

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

val data = sqlContext.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} 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.

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. ($p = 2$ by default.) This normalization can help standardize your input data and improve the behavior of learning algorithms.

The following example demonstrates how to load a dataset in libsvm format and then normalize each row to have unit L^2 norm and unit L^∞ norm.

Scala Java Python

Refer to the Normalizer Scala docs for more details on the API.

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

val dataFrame = sqlContext.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Normalize each Vector using $L^1$ norm.
val normalizer = new Normalizer()
  .setInputCol("features")
  .setOutputCol("normFeatures")
  .setP(1.0)

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

```
l1NormData.show()

// Normalize each Vector using $L^\infty$ norm.
val lInfNormData = normalizer.transform(dataFrame, normalizer.p -> Double.PositiveInfinity)
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 this does not work on sparse input and will raise an exception.

`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.

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

Refer to the StandardScaler Scala docs for more details on the API.

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

val dataFrame = sqlContext.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.

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,

$$Rescaled(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}} * (max - min) + min \tag{1}$$

For the case `E_{max} == E_{min}`, `Rescaled(e_i) = 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.

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

Scala Java

Refer to the MinMaxScaler Scala docs and the MinMaxScalerModel Scala docs for more details on the API.

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

val dataFrame = sqlContext.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

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)
scaledData.show()
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/MinMaxScalerExample.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 bound and lower bound of the targeted column, you would better add the `Double.NegativeInfinity` and `Double.PositiveInfinity` as the bounds of your splits to prevent a potenial 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.

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

Scala **Java** **Python**

Refer to the Bucketizer Scala docs for more details on the API.

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

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

val data = Array(-0.5, -0.3, 0.0, 0.2)
val dataFrame = sqlContext.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)
bucketedData.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.

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \circ \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_N w_N \end{pmatrix}$$

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

Scala **Java** **Python**

Refer to the ElementwiseProduct Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.ElementwiseProduct
import org.apache.spark.mllib.linalg.Vectors

// Create some vector data; also works for sparse vectors
val dataFrame = sqlContext.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, it 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**

Refer to the SQLTransformer Scala docs for more details on the API.

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

val df = sqlContext.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**

«

Refer to the VectorAssembler Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.mllib.linalg.Vectors

val dataset = sqlContext.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(output.select("features", "clicked").first())
```

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

QuantileDiscretizer

`QuantileDiscretizer` takes a column with continuous features and outputs a column with binned categorical features. The bin ranges are chosen by taking a sample of the data and dividing it into roughly equal parts. The lower and upper bin bounds will be `-Infinity` and `+Infinity`, covering all real values. This attempts to find `numBuckets` partitions based on a sample of the given input data, but it may find fewer depending on the data sample values.

Note that the result may be different every time you run it, since the sample strategy behind it is non-deterministic.

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 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**

Refer to the QuantileDiscretizer Scala docs for more details on the API.

```
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 = sc.parallelize(data).toDF("id", "hour")

val discretizer = new QuantileDiscretizer()
  .setInputCol("hour")
  .setOutputCol("result")
  .setNumBuckets(3)
```

«

```
val result = discretizer.fit(df).transform(df)
result.show()
```

Find full example code at "examples/src/main/scala/org/apache/spark/examples/ml/QuantileDiscretizerExample.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 a 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 represents the indices into the vector, `setIndices()`;
2. String indices that represents 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 threw out when encountering with empty input attributes.

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. Assuming that the first column of `userFeatures` are all zeros, so we want to remove it and only the last two columns are selected. 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 a 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**

Refer to the VectorSlicer Scala docs for more details on the API.

```
import org.apache.spark.ml.attribute.{Attribute, AttributeGroup, NumericAttribute}
import org.apache.spark.ml.feature.VectorSlicer
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.StructType

val data = Array(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 dataRDD = sc.parallelize(data)
val dataset = sqlContext.createDataFrame(dataRDD, 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)
println(output.select("userFeatures", "features").first())
```

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. It produces a vector column of features and a double column of labels. Like when formulas are used in R for linear regression, string input columns will be one-hot encoded, and numeric columns will be cast to doubles. If not already present in the DataFrame, the output label column will be created from the specified response variable in the formula.

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

Refer to the RFormula Scala docs for more details on the API.

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

val dataset = sqlContext.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 orders features based on a Chi-Squared test of independence from the class, and then filters (selects) the top features which the class label depends on the most. This is akin to yielding the features with the most predictive power.

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 a `numTopFeatures = 1`, then according to our label `clicked` the last column in our `features` 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**

Refer to the ChiSqSelector Scala docs for more details on the API.

```
import org.apache.spark.ml.feature.ChiSqSelector
import org.apache.spark.mllib.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 = sc.parallelize(data).toDF("id", "features", "clicked")

val selector = new ChiSqSelector()
  .setNumTopFeatures(1)
  .setFeaturesCol("features")
  .setLabelCol("clicked")
```

«

```
  .setOutputCol("selectedFeatures")

val result = selector.fit(df).transform(df)
result.show()
```

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

«