Extracting, transforming and selecting features

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
- Locality Sensitive Hashing (LSH): This class of algorithms combines aspects of feature transformation with other algorithms.

Table of Contents

Feature Extractors

TF-IDF

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 t, a document by d, and the corpus by D. Term frequency $TF(t, d)$ is the number of times that term t appears in document d, while document frequency $DF(t, D)$ is the number of documents that contains term t. 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:

$$IDF(t, D) = \log\frac{|D| + 1}{DF(t, D) + 1},$$

where $|D|$ 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:

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D).$$

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 $2^{18} = 262,144$. 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.

`CountVectorizer` converts text documents to vectors of term counts. Refer to CountVectorizer for more details.

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.

Note: `spark.ml` doesn't provide tools for text segmentation. We refer users to the Stanford NLP Group and scalanlp/chalk.

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

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

```python
from pyspark.ml.feature import HashingTF, IDF, Tokenizer

sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

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

Find full example code at "examples/src/main/python/ml/tf_idf_example.py" 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**

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

```
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or document.
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))
```

Find full example code at "examples/src/main/python/ml/word2vec_example.py" 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**

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

»

```
from pyspark.ml.feature import CountVectorizer

# Input data: Each row is a bag of words with a ID.
df = spark.createDataFrame([
    (0, "a b c".split(" ")),
    (1, "a b b c a".split(" "))
], ["id", "words"])

# fit a CountVectorizerModel from the corpus.
cv = CountVectorizer(inputCol="words", outputCol="features", vocabSize=3, minDF=2.0)

model = cv.fit(df)

result = model.transform(df)
result.show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/count_vectorizer_example.py" 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**

Refer to the FeatureHasher Python docs for more details on the API.

»

```
from pyspark.ml.feature import FeatureHasher

dataset = spark.createDataFrame([
    (2.2, True, "1", "foo"),
    (3.3, False, "2", "bar"),
    (4.4, False, "3", "baz"),
    (5.5, False, "4", "foo")
], ["real", "bool", "stringNum", "string"])

hasher = FeatureHasher(inputCols=["real", "bool", "stringNum", "string"],
                       outputCol="features")

featurized = hasher.transform(dataset)
featurized.show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/feature_hasher_example.py" 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

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

```
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import IntegerType

sentenceDataFrame = spark.createDataFrame([
    (0, "Hi I heard about Spark"),
    (1, "I wish Java could use case classes"),
    (2, "Logistic,regression,models,are,neat")
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words", pattern="\\W")
# alternatively, pattern="\\w+", gaps(False)

countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")\
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)

regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words") \
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/tokenizer_example.py" 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**

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

```python
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/stopwords_remover_example.py" 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.

Examples

Scala **Java** **Python**

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

```python
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/n_gram_example.py" 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**

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

```
from pyspark.ml.feature import Binarizer

continuousDataFrame = spark.createDataFrame([
    (0, 0.1),
    (1, 0.8),
    (2, 0.2)
], ["id", "feature"])

binarizer = Binarizer(threshold=0.5, inputCol="feature", outputCol="binarized_feature")

binarizedDataFrame = binarizer.transform(continuousDataFrame)

print("Binarizer output with Threshold = %f" % binarizer.getThreshold())
binarizedDataFrame.show()
```

Find full example code at "examples/src/main/python/ml/binarizer_example.py" 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

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

```
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors

data = [(Vectors.sparse(5, [(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]),)]
df = spark.createDataFrame(data, ["features"])

pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)

result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/pca_example.py" 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

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

```
from pyspark.ml.feature import PolynomialExpansion
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (Vectors.dense([2.0, 1.0]),),
    (Vectors.dense([0.0, 0.0]),),
    (Vectors.dense([3.0, -1.0]),)
], ["features"])

polyExpansion = PolynomialExpansion(degree=3, inputCol="features", outputCol="polyFeatures")
polyDF = polyExpansion.transform(df)

polyDF.show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/polynomial_expansion_example.py" 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).

Examples

Scala Java Python

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

```
from pyspark.ml.feature import DCT
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (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]),)], ["features"])

dct = DCT(inverse=False, inputCol="features", outputCol="featuresDCT")

dctDf = dct.transform(df)

dctDf.select("featuresDCT").show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/dct_example.py" 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

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

```
from pyspark.ml.feature import StringIndexer

df = spark.createDataFrame(
    [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
    ["id", "category"])

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed = indexer.fit(df).transform(df)
indexed.show()
```

Find full example code at "examples/src/main/python/ml/string_indexer_example.py" 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

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

```python
from pyspark.ml.feature import IndexToString, StringIndexer

df = spark.createDataFrame(
    [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],
    ["id", "category"])

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
model = indexer.fit(df)
indexed = model.transform(df)

print("Transformed string column '%s' to indexed column '%s'"
      % (indexer.getInputCol(), indexer.getOutputCol()))
indexed.show()

print("StringIndexer will store labels in output column metadata\n")

converter = IndexToString(inputCol="categoryIndex", outputCol="originalCategory")
converted = converter.transform(indexed)

print("Transformed indexed column '%s' back to original string column '%s' using "
      "labels in metadata" % (converter.getInputCol(), converter.getOutputCol()))
converted.select("id", "categoryIndex", "originalCategory").show()
```

Find full example code at "examples/src/main/python/ml/index_to_string_example.py" 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

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

```python
from pyspark.ml.feature import OneHotEncoder

df = spark.createDataFrame([
    (0.0, 1.0),
    (1.0, 0.0),
    (2.0, 1.0),
    (0.0, 2.0),
    (0.0, 1.0),
    (2.0, 0.0)
], ["categoryIndex1", "categoryIndex2"])

encoder = OneHotEncoder(inputCols=["categoryIndex1", "categoryIndex2"],
                        outputCols=["categoryVec1", "categoryVec2"])
model = encoder.fit(df)
encoded = model.transform(df)
encoded.show()
```

Find full example code at "examples/src/main/python/ml/onehot_encoder_example.py" 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.

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

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

```python
from pyspark.ml.feature import VectorIndexer

data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

indexer = VectorIndexer(inputCol="features", outputCol="indexed", maxCategories=10)
indexerModel = indexer.fit(data)

categoricalFeatures = indexerModel.categoryMaps
print("Chose %d categorical features: %s" %
      (len(categoricalFeatures), ", ".join(str(k) for k in categoricalFeatures.keys())))

# Create new column "indexed" with categorical values transformed to indices
indexedData = indexerModel.transform(data)
indexedData.show()
```

Find full example code at "examples/src/main/python/ml/vector_indexer_example.py" 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**

Refer to the Interaction Python docs for more details on the API.

```python
from pyspark.ml.feature import Interaction, VectorAssembler

df = spark.createDataFrame(
    [(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)],
    ["id1", "id2", "id3", "id4", "id5", "id6", "id7"])

assembler1 = VectorAssembler(inputCols=["id2", "id3", "id4"], outputCol="vec1")

assembled1 = assembler1.transform(df)

assembler2 = VectorAssembler(inputCols=["id5", "id6", "id7"], outputCol="vec2")

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

interaction = Interaction(inputCols=["id1", "vec1", "vec2"], outputCol="interactedCol")

interacted = interaction.transform(assembled2)

interacted.show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/interaction_example.py" 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.

Examples

The following example demonstrates how to load a dataset in libsvm format and then normalize each row to have unit L^1 norm and unit L^∞ norm.

Scala Java **Python**

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

```python
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (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]),)
], ["id", "features"])

# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normFeatures", p=1.0)
l1NormData = normalizer.transform(dataFrame)
print("Normalized using L^1 norm")
l1NormData.show()

# Normalize each Vector using $L^\infty$ norm.
lInfNormData = normalizer.transform(dataFrame, {normalizer.p: float("inf")})
print("Normalized using L^inf norm")
lInfNormData.show()
```

Find full example code at "examples/src/main/python/ml/normalizer_example.py" 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

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

```python
from pyspark.ml.feature import StandardScaler

dataFrame = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures",
                        withStd=True, withMean=False)

# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(dataFrame)

# Normalize each feature to have unit standard deviation.
scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

Find full example code at "examples/src/main/python/ml/standard_scaler_example.py" 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

» Refer to the RobustScaler Python docs for more details on the API.

```
from pyspark.ml.feature import RobustScaler

dataFrame = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")
scaler = RobustScaler(inputCol="features", outputCol="scaledFeatures",
                      withScaling=True, withCentering=False,
                      lower=0.25, upper=0.75)

# Compute summary statistics by fitting the RobustScaler
scalerModel = scaler.fit(dataFrame)

# Transform each feature to have unit quantile range.
scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

Find full example code at "examples/src/main/python/ml/robust_scaler_example.py" 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,

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

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.

Examples

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

Scala Java Python

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

```
from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (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]),)
], ["id", "features"])

scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")

# Compute summary statistics and generate MinMaxScalerModel
scalerModel = scaler.fit(dataFrame)

# rescale each feature to range [min, max].
scaledData = scalerModel.transform(dataFrame)
print("Features scaled to range: [%f, %f]" % (scaler.getMin(), scaler.getMax()))
scaledData.select("features", "scaledFeatures").show()
```

Find full example code at "examples/src/main/python/ml/min_max_scaler_example.py" 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

Refer to the MaxAbsScaler Python docs and the MaxAbsScalerModel Python docs for more details on the API.

```
from pyspark.ml.feature import MaxAbsScaler
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (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]),)
], ["id", "features"])

scaler = MaxAbsScaler(inputCol="features", outputCol="scaledFeatures")

# Compute summary statistics and generate MaxAbsScalerModel
scalerModel = scaler.fit(dataFrame)

# rescale each feature to range [-1, 1].
scaledData = scalerModel.transform(dataFrame)

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

Find full example code at "examples/src/main/python/ml/max_abs_scaler_example.py" 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 `Double`s into another index-wised column.

Scala Java Python

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

»

```
from pyspark.ml.feature import Bucketizer

splits = [-float("inf"), -0.5, 0.0, 0.5, float("inf")]

data = [(-999.9,), (-0.5,), (-0.3,), (0.0,), (0.2,), (999.9,)]
dataFrame = spark.createDataFrame(data, ["features"])

bucketizer = Bucketizer(splits=splits, inputCol="features", outputCol="bucketedFeatures")

# Transform original data into its bucket index.
bucketedData = bucketizer.transform(dataFrame)

print("Bucketizer output with %d buckets" % (len(bucketizer.getSplits())-1))
bucketedData.show()
```

Find full example code at "examples/src/main/python/ml/bucketizer_example.py" 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.

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

Examples

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

Scala Java **Python**

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

```
from pyspark.ml.feature import ElementwiseProduct
from pyspark.ml.linalg import Vectors

# Create some vector data; also works for sparse vectors
data = [(Vectors.dense([1.0, 2.0, 3.0]),), (Vectors.dense([4.0, 5.0, 6.0]),)]
df = spark.createDataFrame(data, ["vector"])
transformer = ElementwiseProduct(scalingVec=Vectors.dense([0.0, 1.0, 2.0]),
                                 inputCol="vector", outputCol="transformedVector")
# Batch transform the vectors to create new column:
transformer.transform(df).show()
```

Find full example code at "examples/src/main/python/ml/elementwise_product_example.py" 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**

» Refer to the SQLTransformer Python docs for more details on the API.

```python
from pyspark.ml.feature import SQLTransformer

df = spark.createDataFrame([
    (0, 1.0, 3.0),
    (2, 2.0, 5.0)
], ["id", "v1", "v2"])
sqlTrans = SQLTransformer(
    statement="SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__")
sqlTrans.transform(df).show()
```

Find full example code at "examples/src/main/python/ml/sql_transformer.py" 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 Python docs for more details on the API.

```python
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

dataset = spark.createDataFrame(
    [(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)],
    ["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")

output = assembler.transform(dataset)
print("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'")
output.select("features", "clicked").show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/vector_assembler_example.py" 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

Refer to the VectorSizeHint Python docs for more details on the API.

```python
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import (VectorSizeHint, VectorAssembler)

dataset = spark.createDataFrame(
    [(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)],
    ["id", "hour", "mobile", "userFeatures", "clicked"])

sizeHint = VectorSizeHint(
    inputCol="userFeatures",
    handleInvalid="skip",
    size=3)

datasetWithSize = sizeHint.transform(dataset)
print("Rows where 'userFeatures' is not the right size are filtered out")
datasetWithSize.show(truncate=False)

assembler = VectorAssembler(
    inputCols=["hour", "mobile", "userFeatures"],
    outputCol="features")

# This dataframe can be used by downstream transformers as before
output = assembler.transform(datasetWithSize)
print("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column 'features'")
output.select("features", "clicked").show(truncate=False)
```

Find full example code at "examples/src/main/python/ml/vector_size_hint_example.py" 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**

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

```python
from pyspark.ml.feature import QuantileDiscretizer

data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2)]
df = spark.createDataFrame(data, ["id", "hour"])

discretizer = QuantileDiscretizer(numBuckets=3, inputCol="hour", outputCol="result")

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

Find full example code at "examples/src/main/python/ml/quantile_discretizer_example.py" 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**

Refer to the Imputer Python docs for more details on the API.

```
from pyspark.ml.feature import Imputer

df = spark.createDataFrame([
    (1.0, float("nan")),
    (2.0, float("nan")),
    (float("nan"), 3.0),
    (4.0, 4.0),
    (5.0, 5.0)
], ["a", "b"])

imputer = Imputer(inputCols=["a", "b"], outputCols=["out_a", "out_b"])
model = imputer.fit(df)

model.transform(df).show()
```

Find full example code at "examples/src/main/python/ml/imputer_example.py" 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**

» Refer to the VectorSlicer Python docs for more details on the API.

```
from pyspark.ml.feature import VectorSlicer
from pyspark.ml.linalg import Vectors
from pyspark.sql.types import Row

df = spark.createDataFrame([
    Row(userFeatures=Vectors.sparse(3, {0: -2.0, 1: 2.3})),
    Row(userFeatures=Vectors.dense([-2.0, 2.3, 0.0]))])

slicer = VectorSlicer(inputCol="userFeatures", outputCol="features", indices=[1])

output = slicer.transform(df)

output.select("userFeatures", "features").show()
```

Find full example code at "examples/src/main/python/ml/vector_slicer_example.py" 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**

»

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

```
from pyspark.ml.feature import RFormula

dataset = spark.createDataFrame(
    [(7, "US", 18, 1.0),
     (8, "CA", 12, 0.0),
     (9, "NZ", 15, 0.0)],
    ["id", "country", "hour", "clicked"])

formula = RFormula(
    formula="clicked ~ country + hour",
    featuresCol="features",
    labelCol="label")

output = formula.fit(dataset).transform(dataset)
output.select("features", "label").show()
```

Find full example code at "examples/src/main/python/ml/rformula_example.py" 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**

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

»

```
from pyspark.ml.feature import ChiSqSelector
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (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,)], ["id", "features", "clicked"])

selector = ChiSqSelector(numTopFeatures=1, featuresCol="features",
                         outputCol="selectedFeatures", labelCol="clicked")

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

print("ChiSqSelector output with top %d features selected" % selector.getNumTopFeatures())
result.show()
```

Find full example code at "examples/src/main/python/ml/chisq_selector_example.py" 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

Refer to the UnivariateFeatureSelector Python docs for more details on the API.

```
from pyspark.ml.feature import UnivariateFeatureSelector
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (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,)], ["id", "features", "label"])

selector = UnivariateFeatureSelector(featuresCol="features", outputCol="selectedFeatures",
                                     labelCol="label", selectionMode="numTopFeatures")
selector.setFeatureType("continuous").setLabelType("categorical").setSelectionThreshold(1)

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

print("UnivariateFeatureSelector output with top %d features selected using f_classif"
      % selector.getSelectionThreshold())
result.show()
```

Find full example code at "examples/src/main/python/ml/univariate_feature_selector_example.py" 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**

Refer to the VarianceThresholdSelector Python docs for more details on the API.

```
from pyspark.ml.feature import VarianceThresholdSelector
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (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]))], ["id", "features"])

selector = VarianceThresholdSelector(varianceThreshold=8.0, outputCol="selectedFeatures")

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

print("Output: Features with variance lower than %f are removed." %
      selector.getVarianceThreshold())
result.show()
```

Find full example code at "examples/src/main/python/ml/variance_threshold_selector_example.py" 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:

$$\forall p, q \in M, d(p,q) \leq r1 \Rightarrow Pr(h(p)=h(q)) \geq p1 d(p,q) \geq r2 \Rightarrow Pr(h(p)=h(q)) \leq p2$$

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 $d(p,q) \geq r2$) which are hashed into the same bucket, and we define a false negative as a pair of nearby features (with $d(p,q) \leq r1$) 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:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$

Its LSH family projects feature vectors $\mathbf{x}$ onto a random unit vector $\mathbf{v}$ and portions the projected results into hash buckets:

$$h(\mathbf{x}) = \Big\lfloor \frac{\mathbf{x} \cdot \mathbf{v}}{r} \Big\rfloor$$

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**

Refer to the BucketedRandomProjectionLSH Python docs for more details on the API.

```python
from pyspark.ml.feature import BucketedRandomProjectionLSH
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import col

dataA = [(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]),)]
dfA = spark.createDataFrame(dataA, ["id", "features"])

dataB = [(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]),)]
dfB = spark.createDataFrame(dataB, ["id", "features"])

key = Vectors.dense([1.0, 0.0])

brp = BucketedRandomProjectionLSH(inputCol="features", outputCol="hashes", bucketLength=2.0,
                                  numHashTables=3)
model = brp.fit(dfA)

# Feature Transformation
print("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)`
print("Approximately joining dfA and dfB on Euclidean distance smaller than 1.5:")
model.approxSimilarityJoin(dfA, dfB, 1.5, distCol="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)`
print("Approximately searching dfA for 2 nearest neighbors of the key:")
model.approxNearestNeighbors(dfA, key, 2).show()
```

Find full example code at "examples/src/main/python/ml/bucketed_random_projection_lsh_example.py" 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:

$$d(\mathbf{A}, \mathbf{B}) = 1 - \frac{|\mathbf{A} \cap \mathbf{B}|}{|\mathbf{A} \cup \mathbf{B}|}$$

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

$$h(\mathbf{A}) = \min_{a \in \mathbf{A}}(g(a))$$

» 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

Refer to the MinHashLSH Python docs for more details on the API.

```python
from pyspark.ml.feature import MinHashLSH
from pyspark.ml.linalg import Vectors
from pyspark.sql.functions import col

dataA = [(0, Vectors.sparse(6, [0, 1, 2], [1.0, 1.0, 1.0]),),
         (1, Vectors.sparse(6, [2, 3, 4], [1.0, 1.0, 1.0]),),
         (2, Vectors.sparse(6, [0, 2, 4], [1.0, 1.0, 1.0]),)]
dfA = spark.createDataFrame(dataA, ["id", "features"])

dataB = [(3, Vectors.sparse(6, [1, 3, 5], [1.0, 1.0, 1.0]),),
         (4, Vectors.sparse(6, [2, 3, 5], [1.0, 1.0, 1.0]),),
         (5, Vectors.sparse(6, [1, 2, 4], [1.0, 1.0, 1.0]),)]
dfB = spark.createDataFrame(dataB, ["id", "features"])

key = Vectors.sparse(6, [1, 3], [1.0, 1.0])

mh = MinHashLSH(inputCol="features", outputCol="hashes", numHashTables=5)
model = mh.fit(dfA)

# Feature Transformation
print("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)`
print("Approximately joining dfA and dfB on distance smaller than 0.6:")
model.approxSimilarityJoin(dfA, dfB, 0.6, distCol="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.
print("Approximately searching dfA for 2 nearest neighbors of the key:")
model.approxNearestNeighbors(dfA, key, 2).show()
```

Find full example code at "examples/src/main/python/ml/min_hash_lsh_example.py" in the Spark repo.