pyspark.mllib package

pyspark.mllib.classification module

class pyspark.mllib.classification.**LogisticRegressionModel**(*weights*, *intercept*, *numFeatures*, *numClasses*) [source]

Classification model trained using Multinomial/Binary Logistic Regression.

Parameters:
- **weights** – Weights computed for every feature.
- **intercept** – Intercept computed for this model. (Only used in Binary Logistic Regression. In Multinomial Logistic Regression, the intercepts will not bea single value, so the intercepts will be part of the weights.)
- **numFeatures** – The dimension of the features.
- **numClasses** – The number of possible outcomes for k classes classification problem in Multinomial Logistic Regression. By default, it is binary logistic regression so numClasses will be set to 2.

```
>>> data = [
...     LabeledPoint(0.0, [0.0, 1.0]),
...     LabeledPoint(1.0, [1.0, 0.0]),
... ]
>>> lrm = LogisticRegressionWithSGD.train(sc.parallelize(data), iterations=10)
>>> lrm.predict([1.0, 0.0])
1
>>> lrm.predict([0.0, 1.0])
0
>>> lrm.predict(sc.parallelize([[1.0, 0.0], [0.0, 1.0]])).collect()
[1, 0]
>>> lrm.clearThreshold()
>>> lrm.predict([0.0, 1.0])
0.279...
```

```
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(0.0, SparseVector(2, {0: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
... ]
>>> lrm = LogisticRegressionWithSGD.train(sc.parallelize(sparse_data), iterations=10)
```

```
>>> lrm.predict(array([0.0, 1.0]))
1
>>> lrm.predict(array([1.0, 0.0]))
0
>>> lrm.predict(SparseVector(2, {1: 1.0}))
1
>>> lrm.predict(SparseVector(2, {0: 1.0}))
0
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> lrm.save(sc, path)
>>> sameModel = LogisticRegressionModel.load(sc, path)
>>> sameModel.predict(array([0.0, 1.0]))
1
>>> sameModel.predict(SparseVector(2, {0: 1.0}))
0
>>> from shutil import rmtree
>>> try:
...    rmtree(path)
... except:
...    pass
>>> multi_class_data = [
...     LabeledPoint(0.0, [0.0, 1.0, 0.0]),
...     LabeledPoint(1.0, [1.0, 0.0, 0.0]),
...     LabeledPoint(2.0, [0.0, 0.0, 1.0])
... ]
>>> data = sc.parallelize(multi_class_data)
>>> mcm = LogisticRegressionWithLBFGS.train(data, iterations=10, numClasses=3)
>>> mcm.predict([0.0, 0.5, 0.0])
0
>>> mcm.predict([0.8, 0.0, 0.0])
1
>>> mcm.predict([0.0, 0.0, 0.3])
2
```

New in version 0.9.0.

clearThreshold()

Clears the threshold so that *predict* will output raw prediction scores. It is used for binary classification only.

New in version 1.4.0.

intercept

Intercept computed for this model.

New in version 1.0.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.4.0.

numClasses [source]

Number of possible outcomes for k classes classification problem in Multinomial Logistic Regression.

New in version 1.4.0.

numFeatures [source]

Dimension of the features.

New in version 1.4.0.

predict(*x*) [source]

Predict values for a single data point or an RDD of points using the model trained.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save this model to the given path.

New in version 1.4.0.

setThreshold(*value*)

Sets the threshold that separates positive predictions from negative predictions. An example with prediction score greater than or equal to this threshold is identified as a positive, and negative otherwise. It is used for binary classification only.

New in version 1.4.0.

threshold

Returns the threshold (if any) used for converting raw prediction scores into 0/1 predictions. It is used for binary classification only.

New in version 1.4.0.

weights

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.classification.**LogisticRegressionWithSGD** [source]

New in version 0.9.0.

> **Note:** Deprecated in 2.0.0. Use ml.classification.LogisticRegression or LogisticRegressionWithLBFGS.

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *miniBatchFraction=1.0*, *initialWeights=None*, *regParam=0.01*, *regType='l2'*, *intercept=False*, *validateData=True*, *convergenceTol=0.001*) [source]

Train a logistic regression model on the given data.

Parameters:
- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **step** – The step parameter used in SGD. (default: 1.0)
- **miniBatchFraction** – Fraction of data to be used for each SGD iteration. (default: 1.0)
- **initialWeights** – The initial weights. (default: None)
- **regParam** – The regularizer parameter. (default: 0.01)
- **regType** –
  The type of regularizer used for training our model. Supported values:
  - “l1” for using L1 regularization
  - “l2” for using L2 regularization (default)
  - None for no regularization
- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e., whether bias features are activated or not). (default: False)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **convergenceTol** – A condition which decides iteration termination. (default: 0.001)

New in version 0.9.0.

class pyspark.mllib.classification.**LogisticRegressionWithLBFGS** [source]

New in version 1.2.0.

classmethod **train**(*data*, *iterations=100*, *initialWeights=None*, *regParam=0.0*, *regType='l2'*, *intercept=False*, *corrections=10*, *tolerance=1e-06*, *validateData=True*, *numClasses=2*) [source]

Train a logistic regression model on the given data.

Parameters:

- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **initialWeights** – The initial weights. (default: None)
- **regParam** – The regularizer parameter. (default: 0.0)
- **regType** –
  The type of regularizer used for training our model. Supported values:
  - "l1" for using L1 regularization
  - "l2" for using L2 regularization (default)
  - None for no regularization
- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e., whether bias features are activated or not). (default: False)
- **corrections** – The number of corrections used in the LBFGS update. If a known updater is used for binary classification, it calls the ml implementation and this parameter will have no effect. (default: 10)
- **tolerance** – The convergence tolerance of iterations for L-BFGS. (default: 1e-6)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **numClasses** – The number of classes (i.e., outcomes) a label can take in Multinomial Logistic Regression. (default: 2)

```
>>> data = [
...     LabeledPoint(0.0, [0.0, 1.0]),
...     LabeledPoint(1.0, [1.0, 0.0]),
... ]
>>> lrm = LogisticRegressionWithLBFGS.train(sc.parallelize(data), iterations=10)
>>> lrm.predict([1.0, 0.0])
1
>>> lrm.predict([0.0, 1.0])
0
```

New in version 1.2.0.

class pyspark.mllib.classification.**SVMModel**(*weights*, *intercept*) [source]

Model for Support Vector Machines (SVMs).

Parameters:
- **weights** – Weights computed for every feature.
- **intercept** – Intercept computed for this model.

```
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(1.0, [1.0]),
...     LabeledPoint(1.0, [2.0]),
...     LabeledPoint(1.0, [3.0])
... ]
>>> svm = SVMWithSGD.train(sc.parallelize(data), iterations=10)
>>> svm.predict([1.0])
1
>>> svm.predict(sc.parallelize([[1.0]])).collect()
[1]
>>> svm.clearThreshold()
>>> svm.predict(array([1.0]))
1.44...
```

```
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {0: -1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
... ]
>>> svm = SVMWithSGD.train(sc.parallelize(sparse_data), iterations=10)
>>> svm.predict(SparseVector(2, {1: 1.0}))
1
>>> svm.predict(SparseVector(2, {0: -1.0}))
0
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> svm.save(sc, path)
>>> sameModel = SVMModel.load(sc, path)
>>> sameModel.predict(SparseVector(2, {1: 1.0}))
1
>>> sameModel.predict(SparseVector(2, {0: -1.0}))
```

```
0
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except:
...     pass
```

New in version 0.9.0.

clearThreshold()

Clears the threshold so that *predict* will output raw prediction scores. It is used for binary classification only.

New in version 1.4.0.

intercept

Intercept computed for this model.

New in version 1.0.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.4.0.

predict(*x*) [source]

Predict values for a single data point or an RDD of points using the model trained.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save this model to the given path.

New in version 1.4.0.

setThreshold(*value*)

Sets the threshold that separates positive predictions from negative predictions. An example with prediction score greater than or equal to this threshold is identified as a positive, and negative otherwise. It is used for binary classification only.

New in version 1.4.0.

threshold

Returns the threshold (if any) used for converting raw prediction scores into 0/1 predictions. It is used for binary classification only.

New in version 1.4.0.

weights

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.classification.**SVMWithSGD** [source]

New in version 0.9.0.

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *regParam=0.01*, *miniBatchFraction=1.0*, *initialWeights=None*, *regType='l2'*, *intercept=False*, *validateData=True*, *convergenceTol=0.001*) [source]

Train a support vector machine on the given data.

Parameters:

- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **step** – The step parameter used in SGD. (default: 1.0)
- **regParam** – The regularizer parameter. (default: 0.01)
- **miniBatchFraction** – Fraction of data to be used for each SGD iteration. (default: 1.0)
- **initialWeights** – The initial weights. (default: None)
- **regType** –
  The type of regularizer used for training our model. Allowed values:
  - “l1” for using L1 regularization
  - “l2” for using L2 regularization (default)
  - None for no regularization
- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not). (default: False)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **convergenceTol** – A condition which decides iteration termination. (default: 0.001)

New in version 0.9.0.

class pyspark.mllib.classification.**NaiveBayesModel**(*labels*, *pi*, *theta*) [source]

Model for Naive Bayes classifiers.

Parameters:
- **labels** – List of labels.
- **pi** – Log of class priors, whose dimension is C, number of labels.
- **theta** – Log of class conditional probabilities, whose dimension is C-by-D, where D is number of features.

```
>>> data = [
...     LabeledPoint(0.0, [0.0, 0.0]),
...     LabeledPoint(0.0, [0.0, 1.0]),
...     LabeledPoint(1.0, [1.0, 0.0]),
... ]
>>> model = NaiveBayes.train(sc.parallelize(data))
>>> model.predict(array([0.0, 1.0]))
0.0
>>> model.predict(array([1.0, 0.0]))
1.0
>>> model.predict(sc.parallelize([[1.0, 0.0]])).collect()
[1.0]
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {1: 0.0})),
...     LabeledPoint(0.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {0: 1.0}))
... ]
>>> model = NaiveBayes.train(sc.parallelize(sparse_data))
>>> model.predict(SparseVector(2, {1: 1.0}))
0.0
>>> model.predict(SparseVector(2, {0: 1.0}))
1.0
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = NaiveBayesModel.load(sc, path)
>>> sameModel.predict(SparseVector(2, {0: 1.0})) == model.predict(SparseVector(2, {0: 1.0}))
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 0.9.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.4.0.

predict(*x*) [source]

Return the most likely class for a data vector or an RDD of vectors

New in version 0.9.0.

save(*sc*, *path*) [source]

Save this model to the given path.

class pyspark.mllib.classification.**NaiveBayes** [source]

New in version 0.9.0.

classmethod **train**(*data*, *lambda_=1.0*) [source]

Train a Naive Bayes model given an RDD of (label, features) vectors.

This is the Multinomial NB (U{http://tinyurl.com/lsdw6p}) which can handle all kinds of discrete data. For example, by converting documents into TF-IDF vectors, it can be used for document classification. By making every vector a 0-1 vector, it can also be used as Bernoulli NB (U{http://tinyurl.com/p7c96j6}). The input feature values must be nonnegative.

Parameters:
- **data** – RDD of LabeledPoint.
- **lambda** – The smoothing parameter. (default: 1.0)

New in version 0.9.0.

class pyspark.mllib.classification.**StreamingLogisticRegressionWithSGD**(*stepSize=0.1*, *numIterations=50*, *miniBatchFraction=1.0*, *regParam=0.0*, *convergenceTol=0.001*) [source]

Train or predict a logistic regression model on streaming data. Training uses Stochastic Gradient Descent to update the model based on each new batch of incoming data from a DStream.

Each batch of data is assumed to be an RDD of LabeledPoints. The number of data points per batch can vary, but the number of features must be

constant. An initial weight vector must be provided.

Parameters:
- **stepSize** – Step size for each iteration of gradient descent. (default: 0.1)
- **numIterations** – Number of iterations run for each batch of data. (default: 50)
- **miniBatchFraction** – Fraction of each batch of data to use for updates. (default: 1.0)
- **regParam** – L2 Regularization parameter. (default: 0.0)
- **convergenceTol** – Value used to determine when to terminate iterations. (default: 0.001)

New in version 1.5.0.

`latestModel`()

Returns the latest model.

New in version 1.5.0.

`predictOn`(*dstream*)

Use the model to make predictions on batches of data from a DStream.

Returns: DStream containing predictions.

New in version 1.5.0.

`predictOnValues`(*dstream*)

Use the model to make predictions on the values of a DStream and carry over its keys.

Returns: DStream containing the input keys and the predictions as values.

New in version 1.5.0.

`setInitialWeights`(*initialWeights*) [source]

Set the initial value of weights.

This must be set before running trainOn and predictOn.

New in version 1.5.0.

`trainOn`(*dstream*) [source]

Train the model on the incoming dstream.

New in version 1.5.0.

pyspark.mllib.clustering module

class pyspark.mllib.clustering.**BisectingKMeansModel**(*java_model*) [source]

A clustering model derived from the bisecting k-means method.

```
>>> data = array([0.0,0.0, 1.0,1.0, 9.0,8.0, 8.0,9.0]).reshape(4, 2)
>>> bskm = BisectingKMeans()
>>> model = bskm.train(sc.parallelize(data, 2), k=4)
>>> p = array([0.0, 0.0])
>>> model.predict(p)
0
>>> model.k
4
>>> model.computeCost(p)
0.0
```

New in version 2.0.0.

clusterCenters [source]

Get the cluster centers, represented as a list of NumPy arrays.

New in version 2.0.0.

computeCost(*x*) [source]

Return the Bisecting K-means cost (sum of squared distances of points to their nearest center) for this model on the given data. If provided with an RDD of points returns the sum.

Parameters: **point** – A data point (or RDD of points) to compute the cost(s).

New in version 2.0.0.

k [source]

Get the number of clusters

New in version 2.0.0.

predict(*x*) [source]

Find the cluster that each of the points belongs to in this model.

Parameters: **x** – A data point (or RDD of points) to determine cluster index.
Returns: Predicted cluster index or an RDD of predicted cluster indices if the input is an RDD.

New in version 2.0.0.

class pyspark.mllib.clustering.**BisectingKMeans** [source]

A bisecting k-means algorithm based on the paper "A comparison of document clustering techniques" by Steinbach, Karypis, and Kumar, with modification to fit Spark. The algorithm starts from a single cluster that contains all points. Iteratively it finds divisible clusters on the bottom level and bisects each of them using k-means, until there are *k* leaf clusters in total or no leaf clusters are divisible. The bisecting steps of clusters on the same level are grouped together to increase parallelism. If bisecting all divisible clusters on the bottom level would result more than *k* leaf clusters, larger clusters get higher priority.

Based on U{http://glaros.dtc.umn.edu/gkhome/fetch/papers/docclusterKDDTMW00.pdf} Steinbach, Karypis, and Kumar, A comparison of document clustering techniques, KDD Workshop on Text Mining, 2000.

New in version 2.0.0.

classmethod **train**(*rdd*, *k=4*, *maxIterations=20*, *minDivisibleClusterSize=1.0*, *seed=-1888008604*) [source]

Runs the bisecting k-means algorithm return the model.

Parameters:
- **rdd** – Training points as an *RDD* of *Vector* or convertible sequence types.
- **k** – The desired number of leaf clusters. The actual number could be smaller if there are no divisible leaf clusters. (default: 4)
- **maxIterations** – Maximum number of iterations allowed to split clusters. (default: 20)
- **minDivisibleClusterSize** – Minimum number of points (if >= 1.0) or the minimum proportion of points (if < 1.0) of a divisible cluster. (default: 1)
- **seed** – Random seed value for cluster initialization. (default: -1888008604 from classOf[BisectingKMeans].getName.##)

New in version 2.0.0.

class pyspark.mllib.clustering.**KMeansModel**(*centers*) [source]

A clustering model derived from the k-means method.

```
>>> data = array([0.0,0.0, 1.0,1.0, 9.0,8.0, 8.0,9.0]).reshape(4, 2)
>>> model = KMeans.train(
...     sc.parallelize(data), 2, maxIterations=10, initializationMode="random",
...                    seed=50, initializationSteps=5, epsilon=1e-4)
>>> model.predict(array([0.0, 0.0])) == model.predict(array([1.0, 1.0]))
True
>>> model.predict(array([8.0, 9.0])) == model.predict(array([9.0, 8.0]))
True
>>> model.k
2
>>> model.computeCost(sc.parallelize(data))
2.0000000000000004
>>> model = KMeans.train(sc.parallelize(data), 2)
>>> sparse_data = [
...     SparseVector(3, {1: 1.0}),
...     SparseVector(3, {1: 1.1}),
...     SparseVector(3, {2: 1.0}),
...     SparseVector(3, {2: 1.1})
... ]
>>> model = KMeans.train(sc.parallelize(sparse_data), 2, initializationMode="k-means||",
...                                    seed=50, initializationSteps=5, epsilon=1e-4)
>>> model.predict(array([0., 1., 0.])) == model.predict(array([0, 1.1, 0.]))
True
>>> model.predict(array([0., 0., 1.])) == model.predict(array([0, 0, 1.1]))
True
>>> model.predict(sparse_data[0]) == model.predict(sparse_data[1])
True
>>> model.predict(sparse_data[2]) == model.predict(sparse_data[3])
True
>>> isinstance(model.clusterCenters, list)
True
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = KMeansModel.load(sc, path)
>>> sameModel.predict(sparse_data[0]) == model.predict(sparse_data[0])
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

```
>>> data = array([-383.1,-382.9, 28.7,31.2, 366.2,367.3]).reshape(3, 2)
>>> model = KMeans.train(sc.parallelize(data), 3, maxIterations=0,
...     initialModel = KMeansModel([(-1000.0,-1000.0),(5.0,5.0),(1000.0,1000.0)]))
>>> model.clusterCenters
[array([-1000., -1000.]), array([ 5.,  5.]), array([ 1000.,  1000.])]
```

New in version 0.9.0.

clusterCenters [source]

Get the cluster centers, represented as a list of NumPy arrays.

New in version 1.0.0.

computeCost(*rdd*) [source]

Return the K-means cost (sum of squared distances of points to their nearest center) for this model on the given data.

Parameters: **rdd** – The RDD of points to compute the cost on.

New in version 1.4.0.

k [source]

Total number of clusters.

New in version 1.4.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.4.0.

predict(*x*) [source]

Find the cluster that each of the points belongs to in this model.

Parameters: **x** – A data point (or RDD of points) to determine cluster index.

Returns: Predicted cluster index or an RDD of predicted cluster indices if the input is an RDD.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save this model to the given path.

New in version 1.4.0.

class pyspark.mllib.clustering.**KMeans** [source]

New in version 0.9.0.

classmethod **train**(*rdd*, *k*, *maxIterations=100*, *runs=1*, *initializationMode='k-means||'*, *seed=None*, *initializationSteps=5*, *epsilon=0.0001*, *initialModel=None*) [source]

Train a k-means clustering model.

Parameters:
- **rdd** – Training points as an *RDD* of *Vector* or convertible sequence types.
- **k** – Number of clusters to create.
- **maxIterations** – Maximum number of iterations allowed. (default: 100)
- **runs** – This param has no effect since Spark 2.0.0.
- **initializationMode** – The initialization algorithm. This can be either “random” or “k-means||”. (default: “k-means||”)
- **seed** – Random seed value for cluster initialization. Set as None to generate seed based on system time. (default: None)
- **initializationSteps** – Number of steps for the k-means|| initialization mode. This is an advanced setting – the default of 5 is almost always enough. (default: 5)
- **epsilon** – Distance threshold within which a center will be considered to have converged. If all centers move less than this Euclidean distance, iterations are stopped. (default: 1e-4)
- **initialModel** – Initial cluster centers can be provided as a KMeansModel object rather than using the random or k-means|| initializationModel. (default: None)

New in version 0.9.0.

class pyspark.mllib.clustering.**GaussianMixtureModel**(*java_model*) [source]

A clustering model derived from the Gaussian Mixture Model method.

```
>>> from pyspark.mllib.linalg import Vectors, DenseMatrix
>>> from numpy.testing import assert_equal
>>> from shutil import rmtree
>>> import os, tempfile
```

```
>>> clusterdata_1 =  sc.parallelize(array([-0.1,-0.05,-0.01,-0.1,
...                                         0.9,0.8,0.75,0.935,
...                                        -0.83,-0.68,-0.91,-0.76 ]).reshape(6, 2), 2)
>>> model = GaussianMixture.train(clusterdata_1, 3, convergenceTol=0.0001,
...                                maxIterations=50, seed=10)
>>> labels = model.predict(clusterdata_1).collect()
>>> labels[0]==labels[1]
False
>>> labels[1]==labels[2]
False
>>> labels[4]==labels[5]
True
>>> model.predict([-0.1,-0.05])
0
>>> softPredicted = model.predictSoft([-0.1,-0.05])
>>> abs(softPredicted[0] - 1.0) < 0.001
True
>>> abs(softPredicted[1] - 0.0) < 0.001
True
>>> abs(softPredicted[2] - 0.0) < 0.001
True
```

```
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = GaussianMixtureModel.load(sc, path)
>>> assert_equal(model.weights, sameModel.weights)
>>> mus, sigmas = list(
...     zip(*[(g.mu, g.sigma) for g in model.gaussians]))
>>> sameMus, sameSigmas = list(
...     zip(*[(g.mu, g.sigma) for g in sameModel.gaussians]))
>>> mus == sameMus
True
>>> sigmas == sameSigmas
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

```
>>> data =  array([-5.1971, -2.5359, -3.8220,
...                -5.2211, -5.0602,  4.7118,
...                 6.8989, 3.4592,  4.6322,
```

```
...                 5.7048,  4.6567, 5.5026,
...                 4.5605,  5.2043,  6.2734])
>>> clusterdata_2 = sc.parallelize(data.reshape(5,3))
>>> model = GaussianMixture.train(clusterdata_2, 2, convergenceTol=0.0001,
...                               maxIterations=150, seed=10)
>>> labels = model.predict(clusterdata_2).collect()
>>> labels[0]==labels[1]
True
>>> labels[2]==labels[3]==labels[4]
True
```

New in version 1.3.0.

gaussians [source]

Array of MultivariateGaussian where gaussians[i] represents the Multivariate Gaussian (Normal) Distribution for Gaussian i.

New in version 1.4.0.

k [source]

Number of gaussians in mixture.

New in version 1.4.0.

classmethod **load**(*sc*, *path*) [source]

Load the GaussianMixtureModel from disk.

Parameters:
- **sc** – SparkContext.
- **path** – Path to where the model is stored.

New in version 1.5.0.

predict(*x*) [source]

Find the cluster to which the point ‘x’ or each point in RDD ‘x’ has maximum membership in this model.

Parameters: **x** – A feature vector or an RDD of vectors representing data points.

Returns: Predicted cluster label or an RDD of predicted cluster labels if the input is an RDD.

New in version 1.3.0.

predictSoft(*x*) [source]

Find the membership of point ‘x’ or each point in RDD ‘x’ to all mixture components.

Parameters: **x** – A feature vector or an RDD of vectors representing data points.

Returns: The membership value to all mixture components for vector ‘x’ or each vector in RDD ‘x’.

New in version 1.3.0.

weights [source]

Weights for each Gaussian distribution in the mixture, where weights[i] is the weight for Gaussian i, and weights.sum == 1.

New in version 1.4.0.

class pyspark.mllib.clustering.**GaussianMixture** [source]

Learning algorithm for Gaussian Mixtures using the expectation-maximization algorithm.

New in version 1.3.0.

classmethod **train**(*rdd*, *k*, *convergenceTol=0.001*, *maxIterations=100*, *seed=None*, *initialModel=None*) [source]

Train a Gaussian Mixture clustering model.

Parameters:
- **rdd** – Training points as an *RDD* of *Vector* or convertible sequence types.
- **k** – Number of independent Gaussians in the mixture model.
- **convergenceTol** – Maximum change in log-likelihood at which convergence is considered to have occurred. (default: 1e-3)
- **maxIterations** – Maximum number of iterations allowed. (default: 100)
- **seed** – Random seed for initial Gaussian distribution. Set as None to generate seed based on system time. (default: None)
- **initialModel** – Initial GMM starting point, bypassing the random initialization. (default: None)

New in version 1.3.0.

class pyspark.mllib.clustering.**PowerIterationClusteringModel**(*java_model*) [source]

Model produced by [[PowerIterationClustering]].

```
>>> import math
>>> def genCircle(r, n):
...     points = []
...     for i in range(0, n):
```

```
...         theta = 2.0 * math.pi * i / n
...         points.append((r * math.cos(theta), r * math.sin(theta)))
...     return points
>>> def sim(x, y):
...     dist2 = (x[0] - y[0]) * (x[0] - y[0]) + (x[1] - y[1]) * (x[1] - y[1])
...     return math.exp(-dist2 / 2.0)
>>> r1 = 1.0
>>> n1 = 10
>>> r2 = 4.0
>>> n2 = 40
>>> n = n1 + n2
>>> points = genCircle(r1, n1) + genCircle(r2, n2)
>>> similarities = [(i, j, sim(points[i], points[j])) for i in range(1, n) for j in range(0, i)]
>>> rdd = sc.parallelize(similarities, 2)
>>> model = PowerIterationClustering.train(rdd, 2, 40)
>>> model.k
2
>>> result = sorted(model.assignments().collect(), key=lambda x: x.id)
>>> result[0].cluster == result[1].cluster == result[2].cluster == result[3].cluster
True
>>> result[4].cluster == result[5].cluster == result[6].cluster == result[7].cluster
True
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = PowerIterationClusteringModel.load(sc, path)
>>> sameModel.k
2
>>> result = sorted(model.assignments().collect(), key=lambda x: x.id)
>>> result[0].cluster == result[1].cluster == result[2].cluster == result[3].cluster
True
>>> result[4].cluster == result[5].cluster == result[6].cluster == result[7].cluster
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 1.5.0.

assignments() [source]

Returns the cluster assignments of this model.

New in version 1.5.0.

k [source]

Returns the number of clusters.

New in version 1.5.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.5.0.

class pyspark.mllib.clustering.**PowerIterationClustering** [source]

Power Iteration Clustering (PIC), a scalable graph clustering algorithm developed by [[http://www.icml2010.org/papers/387.pdf Lin and Cohen]]. From the abstract: PIC finds a very low-dimensional embedding of a dataset using truncated power iteration on a normalized pair-wise similarity matrix of the data.

New in version 1.5.0.

class **Assignment** [source]

Represents an (id, cluster) tuple.

New in version 1.5.0.

classmethod PowerIterationClustering.**train**(*rdd*, *k*, *maxIterations=100*, *initMode='random'*) [source]

Parameters:
- **rdd** – An RDD of (i, j, s_{ij}) tuples representing the affinity matrix, which is the matrix A in the PIC paper. The similarity s_{ij}must be nonnegative. This is a symmetric matrix and hence s_{ij}= s_{ji} For any (i, j) with nonzero similarity, there should be either (i, j, s_{ij}) or (j, i, s_{ji}) in the input. Tuples with i = j are ignored, because it is assumed s_{ij}= 0.0.
- **k** – Number of clusters.
- **maxIterations** – Maximum number of iterations of the PIC algorithm. (default: 100)
- **initMode** – Initialization mode. This can be either “random” to use a random vector as vertex properties, or “degree” to use normalized sum similarities. (default: “random”)

New in version 1.5.0.

class pyspark.mllib.clustering.**StreamingKMeans**(*k=2*, *decayFactor=1.0*, *timeUnit='batches'*) [source]

Provides methods to set k, decayFactor, timeUnit to configure the KMeans algorithm for fitting and predicting on incoming dstreams. More details on how the centroids are updated are provided under the docs of StreamingKMeansModel.

Parameters:
- **k** – Number of clusters. (default: 2)
- **decayFactor** – Forgetfulness of the previous centroids. (default: 1.0)
- **timeUnit** – Can be “batches” or “points”. If points, then the decay factor is raised to the power of number of new points and if batches, then decay factor will be used as is. (default: “batches”)

New in version 1.5.0.

latestModel() [source]

Return the latest model

New in version 1.5.0.

predictOn(*dstream*) [source]

Make predictions on a dstream. Returns a transformed dstream object

New in version 1.5.0.

predictOnValues(*dstream*) [source]

Make predictions on a keyed dstream. Returns a transformed dstream object.

New in version 1.5.0.

setDecayFactor(*decayFactor*) [source]

Set decay factor.

New in version 1.5.0.

setHalfLife(*halfLife*, *timeUnit*) [source]

Set number of batches after which the centroids of that particular batch has half the weightage.

New in version 1.5.0.

setInitialCenters(*centers*, *weights*) [source]

Set initial centers. Should be set before calling trainOn.

New in version 1.5.0.

setK(*k*) [source]

Set number of clusters.

New in version 1.5.0.

setRandomCenters(*dim*, *weight*, *seed*) [source]

Set the initial centres to be random samples from a gaussian population with constant weights.

New in version 1.5.0.

trainOn(*dstream*) [source]

Train the model on the incoming dstream.

New in version 1.5.0.

class pyspark.mllib.clustering.**StreamingKMeansModel**(*clusterCenters*, *clusterWeights*) [source]

Clustering model which can perform an online update of the centroids.

The update formula for each centroid is given by

- c_t+1 = ((c_t * n_t * a) + (x_t * m_t)) / (n_t + m_t)
- n_t+1 = n_t * a + m_t

where

- c_t: Centroid at the n_th iteration.
- n_t: Number of samples (or) weights associated with the centroid at the n_th iteration.
- x_t: Centroid of the new data closest to c_t.
- m_t: Number of samples (or) weights of the new data closest to c_t

- c_t+1: New centroid.
- n_t+1: New number of weights.
- a: Decay Factor, which gives the forgetfulness.

Note that if a is set to 1, it is the weighted mean of the previous and new data. If it set to zero, the old centroids are completely forgotten.

Parameters:
- **clusterCenters** – Initial cluster centers.
- **clusterWeights** – List of weights assigned to each cluster.

```
>>> initCenters = [[0.0, 0.0], [1.0, 1.0]]
>>> initWeights = [1.0, 1.0]
>>> stkm = StreamingKMeansModel(initCenters, initWeights)
>>> data = sc.parallelize([[-0.1, -0.1], [0.1, 0.1],
...                        [0.9, 0.9], [1.1, 1.1]])
>>> stkm = stkm.update(data, 1.0, u"batches")
>>> stkm.centers
array([[ 0.,  0.],
       [ 1.,  1.]])
>>> stkm.predict([-0.1, -0.1])
0
>>> stkm.predict([0.9, 0.9])
1
>>> stkm.clusterWeights
[3.0, 3.0]
>>> decayFactor = 0.0
>>> data = sc.parallelize([DenseVector([1.5, 1.5]), DenseVector([0.2, 0.2])])
>>> stkm = stkm.update(data, 0.0, u"batches")
>>> stkm.centers
array([[ 0.2,  0.2],
       [ 1.5,  1.5]])
>>> stkm.clusterWeights
[1.0, 1.0]
>>> stkm.predict([0.2, 0.2])
0
>>> stkm.predict([1.5, 1.5])
1
```

New in version 1.5.0.

clusterWeights [source]

Return the cluster weights.

New in version 1.5.0.

update(*data*, *decayFactor*, *timeUnit*) [source]

Update the centroids, according to data

Parameters:
- **data** – RDD with new data for the model update.
- **decayFactor** – Forgetfulness of the previous centroids.
- **timeUnit** – Can be “batches” or “points”. If points, then the decay factor is raised to the power of number of new points and if batches, then decay factor will be used as is.

New in version 1.5.0.

class pyspark.mllib.clustering.**LDA** [source]

New in version 1.5.0.

classmethod **train**(*rdd*, *k=10*, *maxIterations=20*, *docConcentration=-1.0*, *topicConcentration=-1.0*, *seed=None*, *checkpointInterval=10*, *optimizer='em'*) [source]

Train a LDA model.

Parameters:
- **rdd** – RDD of documents, which are tuples of document IDs and term (word) count vectors. The term count vectors are “bags of words” with a fixed-size vocabulary (where the vocabulary size is the length of the vector). Document IDs must be unique and >= 0.
- **k** – Number of topics to infer, i.e., the number of soft cluster centers. (default: 10)
- **maxIterations** – Maximum number of iterations allowed. (default: 20)
- **docConcentration** – Concentration parameter (commonly named “alpha”) for the prior placed on documents’ distributions over topics (“theta”). (default: -1.0)
- **topicConcentration** – Concentration parameter (commonly named “beta” or “eta”) for the prior placed on topics’ distributions over terms. (default: -1.0)
- **seed** – Random seed for cluster initialization. Set as None to generate seed based on system time. (default: None)
- **checkpointInterval** – Period (in iterations) between checkpoints. (default: 10)
- **optimizer** – LDAOptimizer used to perform the actual calculation. Currently “em”, “online” are supported. (default: “em”)

New in version 1.5.0.

class pyspark.mllib.clustering.**LDAModel**(*java_model*) [source]

A clustering model derived from the LDA method.

Latent Dirichlet Allocation (LDA), a topic model designed for text documents. Terminology - “word” = “term”: an element of the vocabulary - “token”: instance of a term appearing in a document - “topic”: multinomial distribution over words representing some concept References: - Original LDA paper (journal version): Blei, Ng, and Jordan. “Latent Dirichlet Allocation.” JMLR, 2003.

```
>>> from pyspark.mllib.linalg import Vectors
>>> from numpy.testing import assert_almost_equal, assert_equal
>>> data = [
...     [1, Vectors.dense([0.0, 1.0])],
...     [2, SparseVector(2, {0: 1.0})],
... ]
>>> rdd =  sc.parallelize(data)
>>> model = LDA.train(rdd, k=2, seed=1)
>>> model.vocabSize()
2
>>> model.describeTopics()
[([1, 0], [0.5..., 0.49...]), ([0, 1], [0.5..., 0.49...])]
>>> model.describeTopics(1)
[([1], [0.5...]), ([0], [0.5...])]
```

```
>>> topics = model.topicsMatrix()
>>> topics_expect = array([[0.5,  0.5], [0.5, 0.5]])
>>> assert_almost_equal(topics, topics_expect, 1)
```

```
>>> import os, tempfile
>>> from shutil import rmtree
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = LDAModel.load(sc, path)
>>> assert_equal(sameModel.topicsMatrix(), model.topicsMatrix())
>>> sameModel.vocabSize() == model.vocabSize()
True
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 1.5.0.

describeTopics(*maxTermsPerTopic=None*) [source]

Return the topics described by weighted terms.

WARNING: If vocabSize and k are large, this can return a large object!

Parameters: **maxTermsPerTopic** – Maximum number of terms to collect for each topic. (default: vocabulary size)

Returns: Array over topics. Each topic is represented as a pair of matching arrays: (term indices, term weights in topic). Each topic's terms are sorted in order of decreasing weight.

New in version 1.6.0.

classmethod **load**(*sc*, *path*) [source]

Load the LDAModel from disk.

Parameters:
- **sc** – SparkContext.
- **path** – Path to where the model is stored.

New in version 1.5.0.

topicsMatrix() [source]

Inferred topics, where each topic is represented by a distribution over terms.

New in version 1.5.0.

vocabSize() [source]

Vocabulary size (number of terms or terms in the vocabulary)

New in version 1.5.0.

pyspark.mllib.evaluation module

class pyspark.mllib.evaluation.**BinaryClassificationMetrics**(*scoreAndLabels*) [source]

Evaluator for binary classification.

Parameters: **scoreAndLabels** – an RDD of (score, label) pairs

```
>>> scoreAndLabels = sc.parallelize([
...     (0.1, 0.0), (0.1, 1.0), (0.4, 0.0), (0.6, 0.0), (0.6, 1.0), (0.6, 1.0), (0.8, 1.0)], 2)
>>> metrics = BinaryClassificationMetrics(scoreAndLabels)
>>> metrics.areaUnderROC
0.70...
>>> metrics.areaUnderPR
0.83...
>>> metrics.unpersist()
```

New in version 1.4.0.

areaUnderPR [source]

Computes the area under the precision-recall curve.

New in version 1.4.0.

areaUnderROC [source]

Computes the area under the receiver operating characteristic (ROC) curve.

New in version 1.4.0.

unpersist() [source]

Unpersists intermediate RDDs used in the computation.

New in version 1.4.0.

class pyspark.mllib.evaluation.**RegressionMetrics**(*predictionAndObservations*) [source]

Evaluator for regression.

Parameters: **predictionAndObservations** – an RDD of (prediction, observation) pairs.

```
>>> predictionAndObservations = sc.parallelize([
...     (2.5, 3.0), (0.0, -0.5), (2.0, 2.0), (8.0, 7.0)])
>>> metrics = RegressionMetrics(predictionAndObservations)
>>> metrics.explainedVariance
8.859...
```

```
>>> metrics.meanAbsoluteError
0.5...
>>> metrics.meanSquaredError
0.37...
>>> metrics.rootMeanSquaredError
0.61...
>>> metrics.r2
0.94...
```

New in version 1.4.0.

explainedVariance [source]

Returns the explained variance regression score. explainedVariance = 1 - variance(y - hat{y}) / variance(y)

New in version 1.4.0.

meanAbsoluteError [source]

Returns the mean absolute error, which is a risk function corresponding to the expected value of the absolute error loss or l1-norm loss.

New in version 1.4.0.

meanSquaredError [source]

Returns the mean squared error, which is a risk function corresponding to the expected value of the squared error loss or quadratic loss.

New in version 1.4.0.

r2 [source]

Returns R^2^, the coefficient of determination.

New in version 1.4.0.

rootMeanSquaredError [source]

Returns the root mean squared error, which is defined as the square root of the mean squared error.

New in version 1.4.0.

class pyspark.mllib.evaluation.**MulticlassMetrics**(*predictionAndLabels*) [source]

Evaluator for multiclass classification.

Parameters: **predictionAndLabels** – an RDD of (prediction, label) pairs.

```
>>> predictionAndLabels = sc.parallelize([(0.0, 0.0), (0.0, 1.0), (0.0, 0.0),
...     (1.0, 0.0), (1.0, 1.0), (1.0, 1.0), (1.0, 1.0), (2.0, 2.0), (2.0, 0.0)])
>>> metrics = MulticlassMetrics(predictionAndLabels)
>>> metrics.confusionMatrix().toArray()
array([[ 2.,  1.,  1.],
       [ 1.,  3.,  0.],
       [ 0.,  0.,  1.]])
>>> metrics.falsePositiveRate(0.0)
0.2...
>>> metrics.precision(1.0)
0.75...
>>> metrics.recall(2.0)
1.0...
>>> metrics.fMeasure(0.0, 2.0)
0.52...
>>> metrics.accuracy
0.66...
>>> metrics.weightedFalsePositiveRate
0.19...
>>> metrics.weightedPrecision
0.68...
>>> metrics.weightedRecall
0.66...
>>> metrics.weightedFMeasure()
0.66...
>>> metrics.weightedFMeasure(2.0)
0.65...
```

New in version 1.4.0.

accuracy [source]

Returns accuracy (equals to the total number of correctly classified instances out of the total number of instances).

New in version 2.0.0.

confusionMatrix() [source]

Returns confusion matrix: predicted classes are in columns, they are ordered by class label ascending, as in “labels”.

New in version 1.4.0.

fMeasure(*label=None*, *beta=None*) [source]

Returns f-measure or f-measure for a given label (category) if specified.

New in version 1.4.0.

falsePositiveRate(*label*) [source]

Returns false positive rate for a given label (category).

New in version 1.4.0.

precision(*label=None*) [source]

Returns precision or precision for a given label (category) if specified.

New in version 1.4.0.

recall(*label=None*) [source]

Returns recall or recall for a given label (category) if specified.

New in version 1.4.0.

truePositiveRate(*label*) [source]

Returns true positive rate for a given label (category).

New in version 1.4.0.

weightedFMeasure(*beta=None*) [source]

Returns weighted averaged f-measure.

New in version 1.4.0.

weightedFalsePositiveRate [source]

Returns weighted false positive rate.

New in version 1.4.0.

weightedPrecision [source]

Returns weighted averaged precision.

New in version 1.4.0.

weightedRecall [source]

Returns weighted averaged recall. (equals to precision, recall and f-measure)

New in version 1.4.0.

weightedTruePositiveRate [source]

Returns weighted true positive rate. (equals to precision, recall and f-measure)

New in version 1.4.0.

class pyspark.mllib.evaluation.**RankingMetrics**(*predictionAndLabels*) [source]

Evaluator for ranking algorithms.

Parameters: **predictionAndLabels** – an RDD of (predicted ranking, ground truth set) pairs.

```
>>> predictionAndLabels = sc.parallelize([
...     ([1, 6, 2, 7, 8, 3, 9, 10, 4, 5], [1, 2, 3, 4, 5]),
...     ([4, 1, 5, 6, 2, 7, 3, 8, 9, 10], [1, 2, 3]),
...     ([1, 2, 3, 4, 5], [])])
>>> metrics = RankingMetrics(predictionAndLabels)
>>> metrics.precisionAt(1)
0.33...
>>> metrics.precisionAt(5)
0.26...
>>> metrics.precisionAt(15)
0.17...
>>> metrics.meanAveragePrecision
0.35...
>>> metrics.ndcgAt(3)
0.33...
>>> metrics.ndcgAt(10)
0.48...
```

New in version 1.4.0.

meanAveragePrecision [source]

Returns the mean average precision (MAP) of all the queries. If a query has an empty ground truth set, the average precision will be zero and a log warining is generated.

New in version 1.4.0.

ndcgAt(*k*) [source]

Compute the average NDCG value of all the queries, truncated at ranking position k. The discounted cumulative gain at position k is computed as: sum,,i=1,,^k^ (2^{relevance of ''i''th item}^ - 1) / log(i + 1), and the NDCG is obtained by dividing the DCG value on the ground truth set. In the current implementation, the relevance value is binary. If a query has an empty ground truth set, zero will be used as NDCG together with a log warning.

New in version 1.4.0.

precisionAt(*k*) [source]

Compute the average precision of all the queries, truncated at ranking position k.

If for a query, the ranking algorithm returns n (n < k) results, the precision value will be computed as #(relevant items retrieved) / k. This formula also applies when the size of the ground truth set is less than k.

If a query has an empty ground truth set, zero will be used as precision together with a log warning.

New in version 1.4.0.

pyspark.mllib.feature module

Python package for feature in MLlib.

class pyspark.mllib.feature.**Normalizer**(*p=2.0*) [source]

Bases: **pyspark.mllib.feature.VectorTransformer**

Normalizes samples individually to unit L^p norm

For any 1 <= *p* < float('inf'), normalizes samples using sum(abs(vector) ^{p}) $^{(1/p)}$ as norm.

For *p* = float('inf'), max(abs(vector)) will be used as norm for normalization.

Parameters: **p** – Normalization in L^p^ space, p = 2 by default.

```
>>> v = Vectors.dense(range(3))
>>> nor = Normalizer(1)
>>> nor.transform(v)
DenseVector([0.0, 0.3333, 0.6667])
```

```
>>> rdd = sc.parallelize([v])
>>> nor.transform(rdd).collect()
[DenseVector([0.0, 0.3333, 0.6667])]
```

```
>>> nor2 = Normalizer(float("inf"))
>>> nor2.transform(v)
DenseVector([0.0, 0.5, 1.0])
```

New in version 1.2.0.

transform(*vector*) [source]

Applies unit length normalization on a vector.

Parameters: **vector** – vector or RDD of vector to be normalized.

Returns: normalized vector. If the norm of the input is zero, it will return the input vector.

New in version 1.2.0.

class pyspark.mllib.feature.**StandardScalerModel**(*java_model*) [source]

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

Represents a StandardScaler model that can transform vectors.

New in version 1.2.0.

mean [source]

Return the column mean values.

New in version 2.0.0.

setWithMean(*withMean*) [source]

Setter of the boolean which decides whether it uses mean or not

New in version 1.4.0.

setWithStd(*withStd*) [source]

Setter of the boolean which decides whether it uses std or not

New in version 1.4.0.

std [source]

Return the column standard deviation values.

New in version 2.0.0.

transform(*vector*) [source]

Applies standardization transformation on a vector.

Note: In Python, transform cannot currently be used within
an RDD transformation or action. Call transform directly on the RDD instead.

Parameters: **vector** – Vector or RDD of Vector to be standardized.
Returns: Standardized vector. If the variance of a column is zero, it will return default *0.0* for the column with zero variance.

New in version 1.2.0.

withMean [source]

Returns if the model centers the data before scaling.

New in version 2.0.0.

withStd [source]

Returns if the model scales the data to unit standard deviation.

New in version 2.0.0.

class pyspark.mllib.feature.**StandardScaler**(*withMean=False*, *withStd=True*) [source]

Bases: **object**

Standardizes features by removing the mean and scaling to unit variance using column summary statistics on the samples in the training set.

Parameters:
- **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.
- **withStd** – True by default. Scales the data to unit standard deviation.

```
>>> vs = [Vectors.dense([-2.0, 2.3, 0]), Vectors.dense([3.8, 0.0, 1.9])]
>>> dataset = sc.parallelize(vs)
>>> standardizer = StandardScaler(True, True)
>>> model = standardizer.fit(dataset)
>>> result = model.transform(dataset)
>>> for r in result.collect(): r
DenseVector([-0.7071, 0.7071, -0.7071])
DenseVector([0.7071, -0.7071, 0.7071])
>>> int(model.std[0])
4
>>> int(model.mean[0]*10)
9
>>> model.withStd
True
>>> model.withMean
True
```

New in version 1.2.0.

fit(*dataset*) [source]

Computes the mean and variance and stores as a model to be used for later scaling.

Parameters: **dataset** – The data used to compute the mean and variance to build the transformation model.

Returns: a StandardScalarModel

New in version 1.2.0.

class pyspark.mllib.feature.**HashingTF**(*numFeatures=1048576*) [source]

Bases: **object**

Maps a sequence of terms to their term frequencies using the hashing trick.

Note: the terms must be hashable (can not be dict/set/list...).

Parameters: **numFeatures** – number of features (default: 2^20)

```
>>> htf = HashingTF(100)
>>> doc = "a a b b c d".split(" ")
>>> htf.transform(doc)
SparseVector(100, {...})
```

New in version 1.2.0.

indexOf(*term*) [source]

Returns the index of the input term.

New in version 1.2.0.

setBinary(*value*) [source]

If True, term frequency vector will be binary such that non-zero term counts will be set to 1 (default: False)

New in version 2.0.0.

transform(*document*) [source]

Transforms the input document (list of terms) to term frequency vectors, or transform the RDD of document to RDD of term frequency vectors.

New in version 1.2.0.

class pyspark.mllib.feature.**IDFModel**(*java_model*) [source]

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

Represents an IDF model that can transform term frequency vectors.

New in version 1.2.0.

idf() [source]

Returns the current IDF vector.

New in version 1.4.0.

transform(*x*) [source]

Transforms term frequency (TF) vectors to TF-IDF vectors.

If *minDocFreq* was set for the IDF calculation, the terms which occur in fewer than *minDocFreq* documents will have an entry of 0.

Note: In Python, transform cannot currently be used within
an RDD transformation or action. Call transform directly on the RDD instead.

Parameters: **x** – an RDD of term frequency vectors or a term frequency vector
Returns: an RDD of TF-IDF vectors or a TF-IDF vector

New in version 1.2.0.

class pyspark.mllib.feature.**IDF**(*minDocFreq=0*) [source]

Bases: **object**

Inverse document frequency (IDF).

The standard formulation is used: *idf = log((m + 1) / (d(t) + 1))*, where *m* is the total number of documents and *d(t)* is the number of documents that contain term *t*.

This implementation supports filtering out terms which do not appear in a minimum number of documents (controlled by the variable *minDocFreq*). For terms that are not in at least *minDocFreq* documents, the IDF is found as 0, resulting in TF-IDFs of 0.

Parameters: **minDocFreq** – minimum of documents in which a term should appear for filtering

```
>>> n = 4
>>> freqs = [Vectors.sparse(n, (1, 3), (1.0, 2.0)),
...          Vectors.dense([0.0, 1.0, 2.0, 3.0]),
...          Vectors.sparse(n, [1], [1.0])]
>>> data = sc.parallelize(freqs)
>>> idf = IDF()
>>> model = idf.fit(data)
```

```
>>> tfidf = model.transform(data)
>>> for r in tfidf.collect(): r
SparseVector(4, {1: 0.0, 3: 0.5754})
DenseVector([0.0, 0.0, 1.3863, 0.863])
SparseVector(4, {1: 0.0})
>>> model.transform(Vectors.dense([0.0, 1.0, 2.0, 3.0]))
DenseVector([0.0, 0.0, 1.3863, 0.863])
>>> model.transform([0.0, 1.0, 2.0, 3.0])
DenseVector([0.0, 0.0, 1.3863, 0.863])
>>> model.transform(Vectors.sparse(n, (1, 3), (1.0, 2.0)))
SparseVector(4, {1: 0.0, 3: 0.5754})
```

New in version 1.2.0.

fit(*dataset*) [source]

Computes the inverse document frequency.

Parameters: **dataset** – an RDD of term frequency vectors

New in version 1.2.0.

class pyspark.mllib.feature.**Word2Vec** [source]

Bases: **object**

Word2Vec creates vector representation of words in a text corpus. The algorithm first constructs a vocabulary from the corpus and then learns vector representation of words in the vocabulary. The vector representation can be used as features in natural language processing and machine learning algorithms.

We used skip-gram model in our implementation and hierarchical softmax method to train the model. The variable names in the implementation matches the original C implementation.

For original C implementation, see https://code.google.com/p/word2vec/ For research papers, see Efficient Estimation of Word Representations in Vector Space and Distributed Representations of Words and Phrases and their Compositionality.

```
>>> sentence = "a b " * 100 + "a c " * 10
>>> localDoc = [sentence, sentence]
>>> doc = sc.parallelize(localDoc).map(lambda line: line.split(" "))
>>> model = Word2Vec().setVectorSize(10).setSeed(42).fit(doc)
```

```
>>> syms = model.findSynonyms("a", 2)
>>> [s[0] for s in syms]
[u'b', u'c']
>>> vec = model.transform("a")
>>> syms = model.findSynonyms(vec, 2)
>>> [s[0] for s in syms]
[u'b', u'c']
```

```
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
>>> sameModel = Word2VecModel.load(sc, path)
>>> model.transform("a") == sameModel.transform("a")
True
>>> syms = sameModel.findSynonyms("a", 2)
>>> [s[0] for s in syms]
[u'b', u'c']
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 1.2.0.

fit(*data*) [source]

Computes the vector representation of each word in vocabulary.

Parameters: **data** – training data. RDD of list of string
Returns: Word2VecModel instance

New in version 1.2.0.

setLearningRate(*learningRate*) [source]

Sets initial learning rate (default: 0.025).

New in version 1.2.0.

setMinCount(*minCount*) [source]

Sets minCount, the minimum number of times a token must appear to be included in the word2vec model's vocabulary (default: 5).

New in version 1.4.0.

setNumIterations(*numIterations*) [source]

Sets number of iterations (default: 1), which should be smaller than or equal to number of partitions.

New in version 1.2.0.

setNumPartitions(*numPartitions*) [source]

Sets number of partitions (default: 1). Use a small number for accuracy.

New in version 1.2.0.

setSeed(*seed*) [source]

Sets random seed.

New in version 1.2.0.

setVectorSize(*vectorSize*) [source]

Sets vector size (default: 100).

New in version 1.2.0.

setWindowSize(*windowSize*) [source]

Sets window size (default: 5).

New in version 2.0.0.

class pyspark.mllib.feature.**Word2VecModel**(*java_model*) [source]

Bases: **pyspark.mllib.feature.JavaVectorTransformer**, **pyspark.mllib.util.JavaSaveable**, **pyspark.mllib.util.JavaLoader**

class for Word2Vec model

New in version 1.2.0.

findSynonyms(*word*, *num*) [source]

Find synonyms of a word

Parameters:
- **word** – a word or a vector representation of word
- **num** – number of synonyms to find

Returns: array of (word, cosineSimilarity)

Note: local use only

New in version 1.2.0.

getVectors() [source]

Returns a map of words to their vector representations.

New in version 1.4.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.5.0.

transform(*word*) [source]

Transforms a word to its vector representation

Note: local use only

Parameters: **word** – a word

Returns: vector representation of word(s)

New in version 1.2.0.

class pyspark.mllib.feature.**ChiSqSelector**(*numTopFeatures*) [source]

Bases: **object**

Creates a ChiSquared feature selector.

Parameters: **numTopFeatures** – number of features that selector will select.

```
>>> data = [
...     LabeledPoint(0.0, SparseVector(3, {0: 8.0, 1: 7.0})),
...     LabeledPoint(1.0, SparseVector(3, {1: 9.0, 2: 6.0})),
...     LabeledPoint(1.0, [0.0, 9.0, 8.0]),
...     LabeledPoint(2.0, [8.0, 9.0, 5.0])
... ]
>>> model = ChiSqSelector(1).fit(sc.parallelize(data))
>>> model.transform(SparseVector(3, {1: 9.0, 2: 6.0}))
SparseVector(1, {0: 6.0})
>>> model.transform(DenseVector([8.0, 9.0, 5.0]))
DenseVector([5.0])
```

New in version 1.4.0.

fit(*data*) [source]

Returns a ChiSquared feature selector.

Parameters: **data** – an *RDD[LabeledPoint]* containing the labeled dataset with categorical features. Real-valued features will be treated as categorical for each distinct value. Apply feature discretizer before using this function.

New in version 1.4.0.

class pyspark.mllib.feature.**ChiSqSelectorModel**(*java_model*) [source]

Bases: **pyspark.mllib.feature.JavaVectorTransformer**

Represents a Chi Squared selector model.

New in version 1.4.0.

transform(*vector*) [source]

Applies transformation on a vector.

Parameters: **vector** – Vector or RDD of Vector to be transformed.

Returns: transformed vector.

New in version 1.4.0.

class pyspark.mllib.feature.**ElementwiseProduct**(*scalingVector*) [source]

Bases: **pyspark.mllib.feature.VectorTransformer**

Scales each column of the vector, with the supplied weight vector. i.e the elementwise product.

```
>>> weight = Vectors.dense([1.0, 2.0, 3.0])
>>> eprod = ElementwiseProduct(weight)
>>> a = Vectors.dense([2.0, 1.0, 3.0])
>>> eprod.transform(a)
DenseVector([2.0, 2.0, 9.0])
>>> b = Vectors.dense([9.0, 3.0, 4.0])
>>> rdd = sc.parallelize([a, b])
>>> eprod.transform(rdd).collect()
[DenseVector([2.0, 2.0, 9.0]), DenseVector([9.0, 6.0, 12.0])]
```

New in version 1.5.0.

transform(*vector*) [source]

Computes the Hadamard product of the vector.

New in version 1.5.0.

pyspark.mllib.fpm module

class pyspark.mllib.fpm.**FPGrowth** [source]

A Parallel FP-growth algorithm to mine frequent itemsets.

New in version 1.4.0.

class **FreqItemset** [source]

Represents an (items, freq) tuple.

New in version 1.4.0.

classmethod FPGrowth.**train**(*data*, *minSupport=0.3*, *numPartitions=-1*) [source]

Computes an FP-Growth model that contains frequent itemsets.

Parameters:
- **data** – The input data set, each element contains a transaction.
- **minSupport** – The minimal support level. (default: 0.3)
- **numPartitions** – The number of partitions used by parallel FP-growth. A value of -1 will use the same number as input data. (default: -1)

New in version 1.4.0.

class pyspark.mllib.fpm.**FPGrowthModel**(*java_model*) [source]

A FP-Growth model for mining frequent itemsets using the Parallel FP-Growth algorithm.

```
>>> data = [["a", "b", "c"], ["a", "b", "d", "e"], ["a", "c", "e"], ["a", "c", "f"]]
>>> rdd = sc.parallelize(data, 2)
>>> model = FPGrowth.train(rdd, 0.6, 2)
>>> sorted(model.freqItemsets().collect())
[FreqItemset(items=[u'a'], freq=4), FreqItemset(items=[u'c'], freq=3), ...
>>> model_path = temp_path + "/fpm"
>>> model.save(sc, model_path)
>>> sameModel = FPGrowthModel.load(sc, model_path)
>>> sorted(model.freqItemsets().collect()) == sorted(sameModel.freqItemsets().collect())
True
```

New in version 1.4.0.

freqItemsets() [source]

Returns the frequent itemsets of this model.

New in version 1.4.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 2.0.0.

class pyspark.mllib.fpm.**PrefixSpan** [source]

A parallel PrefixSpan algorithm to mine frequent sequential patterns. The PrefixSpan algorithm is described in J. Pei, et al., PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth ([[http://doi.org/10.1109/ICDE.2001.914830]]).

New in version 1.6.0.

class **FreqSequence** [source]

Represents a (sequence, freq) tuple.

New in version 1.6.0.

classmethod PrefixSpan.**train**(*data*, *minSupport=0.1*, *maxPatternLength=10*, *maxLocalProjDBSize=32000000*) [source]

Finds the complete set of frequent sequential patterns in the input sequences of itemsets.

Parameters:
- **data** – The input data set, each element contains a sequence of itemsets.
- **minSupport** – The minimal support level of the sequential pattern, any pattern that appears more than (minSupport * size-of-the-dataset) times will be output. (default: 0.1)
- **maxPatternLength** – The maximal length of the sequential pattern, any pattern that appears less than maxPatternLength will be output. (default: 10)
- **maxLocalProjDBSize** – The maximum number of items (including delimiters used in the internal storage format) allowed in a projected database before local processing. If a projected database exceeds this size, another iteration of distributed prefix growth is run. (default: 32000000)

New in version 1.6.0.

class pyspark.mllib.fpm.**PrefixSpanModel**(*java_model*) [source]

Model fitted by PrefixSpan

```
>>> data = [
...     [["a", "b"], ["c"]],
...     [["a"], ["c", "b"], ["a", "b"]],
...     [["a", "b"], ["e"]],
...     [["f"]]]
>>> rdd = sc.parallelize(data, 2)
>>> model = PrefixSpan.train(rdd)
>>> sorted(model.freqSequences().collect())
[FreqSequence(sequence=[[u'a']], freq=3), FreqSequence(sequence=[[u'a'], [u'a']], freq=1), ...
```

New in version 1.6.0.

freqSequences() [source]

Gets frequent sequences

New in version 1.6.0.

pyspark.mllib.linalg module

MLlib utilities for linear algebra. For dense vectors, MLlib uses the NumPy **array** type, so you can simply pass NumPy arrays around. For sparse vectors, users can construct a **SparseVector** object from MLlib or pass SciPy **scipy.sparse** column vectors if SciPy is available in their environment.

class pyspark.mllib.linalg.**Vector** [source]

Bases: **object**

asML() [source]

Convert this vector to the new mllib-local representation. This does NOT copy the data; it copies references.

Returns: **pyspark.ml.linalg.Vector**

toArray() [source]

Convert the vector into an numpy.ndarray

Returns: numpy.ndarray

class pyspark.mllib.linalg.**DenseVector**(*ar*) [source]

Bases: **pyspark.mllib.linalg.Vector**

A dense vector represented by a value array. We use numpy array for storage and arithmetics will be delegated to the underlying numpy array.

```
>>> v = Vectors.dense([1.0, 2.0])
>>> u = Vectors.dense([3.0, 4.0])
>>> v + u
DenseVector([4.0, 6.0])
>>> 2 - v
DenseVector([1.0, 0.0])
>>> v / 2
DenseVector([0.5, 1.0])
>>> v * u
DenseVector([3.0, 8.0])
>>> u / v
DenseVector([3.0, 2.0])
>>> u % 2
```

```
DenseVector([1.0, 0.0])
```

asML() [source]

Convert this vector to the new mllib-local representation. This does NOT copy the data; it copies references.

Returns: **pyspark.ml.linalg.DenseVector**

New in version 2.0.0.

dot(*other*) [source]

Compute the dot product of two Vectors. We support (Numpy array, list, SparseVector, or SciPy sparse) and a target NumPy array that is either 1- or 2-dimensional. Equivalent to calling numpy.dot of the two vectors.

```
>>> dense = DenseVector(array.array('d', [1., 2.]))
>>> dense.dot(dense)
5.0
>>> dense.dot(SparseVector(2, [0, 1], [2., 1.]))
4.0
>>> dense.dot(range(1, 3))
5.0
>>> dense.dot(np.array(range(1, 3)))
5.0
>>> dense.dot([1.,])
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> dense.dot(np.reshape([1., 2., 3., 4.], (2, 2), order='F'))
array([  5.,  11.])
>>> dense.dot(np.reshape([1., 2., 3.], (3, 1), order='F'))
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
```

norm(*p*) [source]

Calculates the norm of a DenseVector.

```
>>> a = DenseVector([0, -1, 2, -3])
>>> a.norm(2)
3.7...
```

```
>>> a.norm(1)
6.0
```

numNonzeros() [source]

Number of nonzero elements. This scans all active values and count non zeros

static **parse**(*s*) [source]

Parse string representation back into the DenseVector.

```
>>> DenseVector.parse(' [ 0.0,1.0,2.0,  3.0]')
DenseVector([0.0, 1.0, 2.0, 3.0])
```

squared_distance(*other*) [source]

Squared distance of two Vectors.

```
>>> dense1 = DenseVector(array.array('d', [1., 2.]))
>>> dense1.squared_distance(dense1)
0.0
>>> dense2 = np.array([2., 1.])
>>> dense1.squared_distance(dense2)
2.0
>>> dense3 = [2., 1.]
>>> dense1.squared_distance(dense3)
2.0
>>> sparse1 = SparseVector(2, [0, 1], [2., 1.])
>>> dense1.squared_distance(sparse1)
2.0
>>> dense1.squared_distance([1.,])
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> dense1.squared_distance(SparseVector(1, [0,], [1.,]))
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
```

toArray() [source]

Returns an numpy.ndarray

values [source]

Returns a list of values

class pyspark.mllib.linalg.**SparseVector**(*size*, **args*) [source]

Bases: **pyspark.mllib.linalg.Vector**

A simple sparse vector class for passing data to MLlib. Users may alternatively pass SciPy's {scipy.sparse} data types.

asML() [source]

Convert this vector to the new mllib-local representation. This does NOT copy the data; it copies references.

Returns: **pyspark.ml.linalg.SparseVector**

New in version 2.0.0.

dot(*other*) [source]

Dot product with a SparseVector or 1- or 2-dimensional Numpy array.

```
>>> a = SparseVector(4, [1, 3], [3.0, 4.0])
>>> a.dot(a)
25.0
>>> a.dot(array.array('d', [1., 2., 3., 4.]))
22.0
>>> b = SparseVector(4, [2], [1.0])
>>> a.dot(b)
0.0
>>> a.dot(np.array([[1, 1], [2, 2], [3, 3], [4, 4]]))
array([ 22.,  22.])
>>> a.dot([1., 2., 3.])
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> a.dot(np.array([1., 2.]))
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> a.dot(DenseVector([1., 2.]))
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> a.dot(np.zeros((3, 2)))
```

```
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
```

indices = *None*

A list of indices corresponding to active entries.

norm(*p*) [source]

Calculates the norm of a SparseVector.

```
>>> a = SparseVector(4, [0, 1], [3., -4.])
>>> a.norm(1)
7.0
>>> a.norm(2)
5.0
```

numNonzeros() [source]

Number of nonzero elements. This scans all active values and count non zeros.

static **parse**(*s*) [source]

Parse string representation back into the SparseVector.

```
>>> SparseVector.parse(' (4, [0,1 ],[ 4.0,5.0] )')
SparseVector(4, {0: 4.0, 1: 5.0})
```

size = *None*

Size of the vector.

squared_distance(*other*) [source]

Squared distance from a SparseVector or 1-dimensional NumPy array.

```
>>> a = SparseVector(4, [1, 3], [3.0, 4.0])
>>> a.squared_distance(a)
0.0
>>> a.squared_distance(array.array('d', [1., 2., 3., 4.]))
```

```
11.0
>>> a.squared_distance(np.array([1., 2., 3., 4.]))
11.0
>>> b = SparseVector(4, [2], [1.0])
>>> a.squared_distance(b)
26.0
>>> b.squared_distance(a)
26.0
>>> b.squared_distance([1., 2.])
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
>>> b.squared_distance(SparseVector(3, [1,], [1.0,]))
Traceback (most recent call last):
    ...
AssertionError: dimension mismatch
```

toArray() [source]

Returns a copy of this SparseVector as a 1-dimensional NumPy array.

values = *None*

A list of values corresponding to active entries.

class pyspark.mllib.linalg.**Vectors** [source]

Bases: **object**

Factory methods for working with vectors. Note that dense vectors are simply represented as NumPy array objects, so there is no need to covert them for use in MLlib. For sparse vectors, the factory methods in this class create an MLlib-compatible type, or users can pass in SciPy's **scipy.sparse** column vectors.

static **dense**(**elements*) [source]

Create a dense vector of 64-bit floats from a Python list or numbers.

```
>>> Vectors.dense([1, 2, 3])
DenseVector([1.0, 2.0, 3.0])
>>> Vectors.dense(1.0, 2.0)
DenseVector([1.0, 2.0])
```

static **fromML**(*vec*) [source]

Convert a vector from the new mllib-local representation. This does NOT copy the data; it copies references.

Parameters: **vec** – a `pyspark.ml.linalg.Vector`

Returns: a `pyspark.mllib.linalg.Vector`

New in version 2.0.0.

static **norm**(*vector*, *p*) [source]

Find norm of the given vector.

static **parse**(*s*) [source]

Parse a string representation back into the Vector.

```
>>> Vectors.parse('[2,1,2 ]')
DenseVector([2.0, 1.0, 2.0])
>>> Vectors.parse(' ( 100,  [0],  [2])')
SparseVector(100, {0: 2.0})
```

static **sparse**(*size*, **args*) [source]

Create a sparse vector, using either a dictionary, a list of (index, value) pairs, or two separate arrays of indices and values (sorted by index).

Parameters:
- **size** – Size of the vector.
- **args** – Non-zero entries, as a dictionary, list of tuples, or two sorted lists containing indices and values.

```
>>> Vectors.sparse(4, {1: 1.0, 3: 5.5})
SparseVector(4, {1: 1.0, 3: 5.5})
>>> Vectors.sparse(4, [(1, 1.0), (3, 5.5)])
SparseVector(4, {1: 1.0, 3: 5.5})
>>> Vectors.sparse(4, [1, 3], [1.0, 5.5])
SparseVector(4, {1: 1.0, 3: 5.5})
```

static **squared_distance**(*v1*, *v2*) [source]

Squared distance between two vectors. a and b can be of type SparseVector, DenseVector, np.ndarray or array.array.

```
>>> a = Vectors.sparse(4, [(0, 1), (3, 4)])
```

```
>>> b = Vectors.dense([2, 5, 4, 1])
>>> a.squared_distance(b)
51.0
```

static **stringify**(*vector*) [source]

Converts a vector into a string, which can be recognized by Vectors.parse().

```
>>> Vectors.stringify(Vectors.sparse(2, [1], [1.0]))
'(2,[1],[1.0])'
>>> Vectors.stringify(Vectors.dense([0.0, 1.0]))
'[0.0,1.0]'
```

static **zeros**(*size*) [source]

class pyspark.mllib.linalg.**Matrix**(*numRows*, *numCols*, *isTransposed=False*) [source]

Bases: **object**

asML() [source]

Convert this matrix to the new mllib-local representation. This does NOT copy the data; it copies references.

toArray() [source]

Returns its elements in a NumPy ndarray.

class pyspark.mllib.linalg.**DenseMatrix**(*numRows*, *numCols*, *values*, *isTransposed=False*) [source]

Bases: **pyspark.mllib.linalg.Matrix**

Column-major dense matrix.

asML() [source]

Convert this matrix to the new mllib-local representation. This does NOT copy the data; it copies references.

Returns: **pyspark.ml.linalg.DenseMatrix**

New in version 2.0.0.

toArray() [source]

Return an numpy.ndarray

```
>>> m = DenseMatrix(2, 2, range(4))
>>> m.toArray()
array([[ 0.,  2.],
       [ 1.,  3.]])
```

toSparse() [source]

Convert to SparseMatrix

class pyspark.mllib.linalg.**SparseMatrix**(*numRows*, *numCols*, *colPtrs*, *rowIndices*, *values*, *isTransposed=False*) [source]

Bases: **pyspark.mllib.linalg.Matrix**

Sparse Matrix stored in CSC format.

asML() [source]

Convert this matrix to the new mllib-local representation. This does NOT copy the data; it copies references.

Returns: **pyspark.ml.linalg.SparseMatrix**

New in version 2.0.0.

toArray() [source]

Return an numpy.ndarray

toDense() [source]

class pyspark.mllib.linalg.**Matrices** [source]

Bases: **object**

static **dense**(*numRows*, *numCols*, *values*) [source]

Create a DenseMatrix

static **fromML**(*mat*) [source]

Convert a matrix from the new mllib-local representation. This does NOT copy the data; it copies references.

Parameters: **mat** – a `pyspark.ml.linalg.Matrix`
Returns: a `pyspark.mllib.linalg.Matrix`

New in version 2.0.0.

static **sparse**(*numRows*, *numCols*, *colPtrs*, *rowIndices*, *values*) [source]

Create a SparseMatrix

class pyspark.mllib.linalg.**QRDecomposition**(*Q*, *R*) [source]

Bases: **object**

Represents QR factors.

Q [source]

An orthogonal matrix Q in a QR decomposition. May be null if not computed.

New in version 2.0.0.

R [source]

An upper triangular matrix R in a QR decomposition.

New in version 2.0.0.

pyspark.mllib.linalg.distributed module

Package for distributed linear algebra.

class pyspark.mllib.linalg.distributed.**DistributedMatrix** [source]

Bases: **object**

Represents a distributively stored matrix backed by one or more RDDs.

numCols() [source]

Get or compute the number of cols.

numRows() [source]

Get or compute the number of rows.

class pyspark.mllib.linalg.distributed.**RowMatrix**(*rows*, *numRows=0*, *numCols=0*) [source]

Bases: **pyspark.mllib.linalg.distributed.DistributedMatrix**

Represents a row-oriented distributed Matrix with no meaningful row indices.

Parameters:
- **rows** – An RDD of vectors.
- **numRows** – Number of rows in the matrix. A non-positive value means unknown, at which point the number of rows will be determined by the number of records in the *rows* RDD.
- **numCols** – Number of columns in the matrix. A non-positive value means unknown, at which point the number of columns will be determined by the size of the first row.

columnSimilarities(*threshold=0.0*) [source]

Compute similarities between columns of this matrix.

The threshold parameter is a trade-off knob between estimate quality and computational cost.

The default threshold setting of 0 guarantees deterministically correct results, but uses the brute-force approach of computing normalized dot products.

Setting the threshold to positive values uses a sampling approach and incurs strictly less computational cost than the brute-force approach. However the similarities computed will be estimates.

The sampling guarantees relative-error correctness for those pairs of columns that have similarity greater than the given similarity threshold.

To describe the guarantee, we set some notation:

- Let A be the smallest in magnitude non-zero element of this matrix.
- Let B be the largest in magnitude non-zero element of this matrix.
- Let L be the maximum number of non-zeros per row.

For example, for {0,1} matrices: A=B=1. Another example, for the Netflix matrix: A=1, B=5

For those column pairs that are above the threshold, the computed similarity is correct to within 20% relative error with probability at least 1 - (0.981)^10/B^

The shuffle size is bounded by the *smaller* of the following two expressions:

- O(n log(n) L / (threshold * A))
- O(m L^2^)

The latter is the cost of the brute-force approach, so for non-zero thresholds, the cost is always cheaper than the brute-force approach.

Param: threshold: Set to 0 for deterministic guaranteed correctness. Similarities above this threshold are estimated with the cost vs estimate quality trade-off described above.

Returns: An n x n sparse upper-triangular CoordinateMatrix of cosine similarities between columns of this matrix.

```
>>> rows = sc.parallelize([[1, 2], [1, 5]])
>>> mat = RowMatrix(rows)
```

```
>>> sims = mat.columnSimilarities()
>>> sims.entries.first().value
0.91914503...
```

New in version 2.0.0.

computeColumnSummaryStatistics() [source]

Computes column-wise summary statistics.

Returns: **MultivariateStatisticalSummary** object containing column-wise summary statistics.

```
>>> rows = sc.parallelize([[1, 2, 3], [4, 5, 6]])
>>> mat = RowMatrix(rows)
```

```
>>> colStats = mat.computeColumnSummaryStatistics()
>>> colStats.mean()
array([ 2.5,  3.5,  4.5])
```

New in version 2.0.0.

computeCovariance() [source]

Computes the covariance matrix, treating each row as an observation. Note that this cannot be computed on matrices with more than 65535

columns.

```
>>> rows = sc.parallelize([[1, 2], [2, 1]])
>>> mat = RowMatrix(rows)
```

```
>>> mat.computeCovariance()
DenseMatrix(2, 2, [0.5, -0.5, -0.5, 0.5], 0)
```

New in version 2.0.0.

computeGramianMatrix() [source]

Computes the Gramian matrix *A^T A*. Note that this cannot be computed on matrices with more than 65535 columns.

```
>>> rows = sc.parallelize([[1, 2, 3], [4, 5, 6]])
>>> mat = RowMatrix(rows)
```

```
>>> mat.computeGramianMatrix()
DenseMatrix(3, 3, [17.0, 22.0, 27.0, 22.0, 29.0, 36.0, 27.0, 36.0, 45.0], 0)
```

New in version 2.0.0.

numCols() [source]

Get or compute the number of cols.

```
>>> rows = sc.parallelize([[1, 2, 3], [4, 5, 6],
...                        [7, 8, 9], [10, 11, 12]])
```

```
>>> mat = RowMatrix(rows)
>>> print(mat.numCols())
3
```

```
>>> mat = RowMatrix(rows, 7, 6)
```

```
>>> print(mat.numCols())
6
```

numRows() [source]

Get or compute the number of rows.

```
>>> rows = sc.parallelize([[1, 2, 3], [4, 5, 6],
...                        [7, 8, 9], [10, 11, 12]])
```

```
>>> mat = RowMatrix(rows)
>>> print(mat.numRows())
4
```

```
>>> mat = RowMatrix(rows, 7, 6)
>>> print(mat.numRows())
7
```

rows [source]

Rows of the RowMatrix stored as an RDD of vectors.

```
>>> mat = RowMatrix(sc.parallelize([[1, 2, 3], [4, 5, 6]]))
>>> rows = mat.rows
>>> rows.first()
DenseVector([1.0, 2.0, 3.0])
```

tallSkinnyQR(*computeQ=False*) [source]

Compute the QR decomposition of this RowMatrix.

The implementation is designed to optimize the QR decomposition (factorization) for the RowMatrix of a tall and skinny shape.

Reference:

Paul G. Constantine, David F. Gleich. “Tall and skinny QR factorizations in MapReduce architectures” ([[http://dx.doi.org/10.1145/1996092.1996103]])

Param: computeQ: whether to computeQ

Returns: QRDecomposition(Q: RowMatrix, R: Matrix), where Q = None if computeQ = false.

```
>>> rows = sc.parallelize([[3, -6], [4, -8], [0, 1]])
>>> mat = RowMatrix(rows)
>>> decomp = mat.tallSkinnyQR(True)
>>> Q = decomp.Q
>>> R = decomp.R
```

```
>>> # Test with absolute values
>>> absQRows = Q.rows.map(lambda row: abs(row.toArray()).tolist())
>>> absQRows.collect()
[[0.6..., 0.0], [0.8..., 0.0], [0.0, 1.0]]
```

```
>>> # Test with absolute values
>>> abs(R.toArray()).tolist()
[[5.0, 10.0], [0.0, 1.0]]
```

New in version 2.0.0.

class pyspark.mllib.linalg.distributed.**IndexedRow**(*index*, *vector*) [source]

Bases: **object**

Represents a row of an IndexedRowMatrix.

Just a wrapper over a (long, vector) tuple.

Parameters:
- **index** – The index for the given row.
- **vector** – The row in the matrix at the given index.

class pyspark.mllib.linalg.distributed.**IndexedRowMatrix**(*rows*, *numRows=0*, *numCols=0*) [source]

Bases: **pyspark.mllib.linalg.distributed.DistributedMatrix**

Represents a row-oriented distributed Matrix with indexed rows.

Parameters:
- **rows** – An RDD of IndexedRows or (long, vector) tuples.

- **numRows** – Number of rows in the matrix. A non-positive value means unknown, at which point the number of rows will be determined by the max row index plus one.
- **numCols** – Number of columns in the matrix. A non-positive value means unknown, at which point the number of columns will be determined by the size of the first row.

columnSimilarities() [source]

Compute all cosine similarities between columns.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(6, [4, 5, 6])])
>>> mat = IndexedRowMatrix(rows)
>>> cs = mat.columnSimilarities()
>>> print(cs.numCols())
3
```

computeGramianMatrix() [source]

Computes the Gramian matrix *A^T A*. Note that this cannot be computed on matrices with more than 65535 columns.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(1, [4, 5, 6])])
>>> mat = IndexedRowMatrix(rows)
```

```
>>> mat.computeGramianMatrix()
DenseMatrix(3, 3, [17.0, 22.0, 27.0, 22.0, 29.0, 36.0, 27.0, 36.0, 45.0], 0)
```

New in version 2.0.0.

numCols() [source]

Get or compute the number of cols.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(1, [4, 5, 6]),
...                        IndexedRow(2, [7, 8, 9]),
...                        IndexedRow(3, [10, 11, 12])])
```

```
>>> mat = IndexedRowMatrix(rows)
>>> print(mat.numCols())
3
```

```
>>> mat = IndexedRowMatrix(rows, 7, 6)
>>> print(mat.numCols())
6
```

numRows() [source]

Get or compute the number of rows.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(1, [4, 5, 6]),
...                        IndexedRow(2, [7, 8, 9]),
...                        IndexedRow(3, [10, 11, 12])])
```

```
>>> mat = IndexedRowMatrix(rows)
>>> print(mat.numRows())
4
```

```
>>> mat = IndexedRowMatrix(rows, 7, 6)
>>> print(mat.numRows())
7
```

rows [source]

Rows of the IndexedRowMatrix stored as an RDD of IndexedRows.

```
>>> mat = IndexedRowMatrix(sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                                        IndexedRow(1, [4, 5, 6])]))
>>> rows = mat.rows
>>> rows.first()
IndexedRow(0, [1.0,2.0,3.0])
```

toBlockMatrix(*rowsPerBlock=1024*, *colsPerBlock=1024*) [source]

Convert this matrix to a BlockMatrix.

Parameters:
- **rowsPerBlock** – Number of rows that make up each block. The blocks forming the final rows are not required to have the given number of rows.
- **colsPerBlock** – Number of columns that make up each block. The blocks forming the final columns are not required to have the given number of columns.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(6, [4, 5, 6])])
>>> mat = IndexedRowMatrix(rows).toBlockMatrix()
```

```
>>> # This IndexedRowMatrix will have 7 effective rows, due to
>>> # the highest row index being 6, and the ensuing
>>> # BlockMatrix will have 7 rows as well.
>>> print(mat.numRows())
7
```

```
>>> print(mat.numCols())
3
```

toCoordinateMatrix() [source]

Convert this matrix to a CoordinateMatrix.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 0]),
...                        IndexedRow(6, [0, 5])])
>>> mat = IndexedRowMatrix(rows).toCoordinateMatrix()
>>> mat.entries.take(3)
[MatrixEntry(0, 0, 1.0), MatrixEntry(0, 1, 0.0), MatrixEntry(6, 0, 0.0)]
```

toRowMatrix() [source]

Convert this matrix to a RowMatrix.

```
>>> rows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
...                        IndexedRow(6, [4, 5, 6])])
```

```
>>> mat = IndexedRowMatrix(rows).toRowMatrix()
>>> mat.rows.collect()
[DenseVector([1.0, 2.0, 3.0]), DenseVector([4.0, 5.0, 6.0])]
```

class pyspark.mllib.linalg.distributed.**MatrixEntry**(*i*, *j*, *value*) [source]

Bases: **object**

Represents an entry of a CoordinateMatrix.

Just a wrapper over a (long, long, float) tuple.

Parameters:
- **i** – The row index of the matrix.
- **j** – The column index of the matrix.
- **value** – The (i, j)th entry of the matrix, as a float.

class pyspark.mllib.linalg.distributed.**CoordinateMatrix**(*entries*, *numRows=0*, *numCols=0*) [source]

Bases: **pyspark.mllib.linalg.distributed.DistributedMatrix**

Represents a matrix in coordinate format.

Parameters:
- **entries** – An RDD of MatrixEntry inputs or (long, long, float) tuples.
- **numRows** – Number of rows in the matrix. A non-positive value means unknown, at which point the number of rows will be determined by the max row index plus one.
- **numCols** – Number of columns in the matrix. A non-positive value means unknown, at which point the number of columns will be determined by the max row index plus one.

entries [source]

Entries of the CoordinateMatrix stored as an RDD of MatrixEntries.

```
>>> mat = CoordinateMatrix(sc.parallelize([MatrixEntry(0, 0, 1.2),
...                                        MatrixEntry(6, 4, 2.1)]))
>>> entries = mat.entries
>>> entries.first()
MatrixEntry(0, 0, 1.2)
```

numCols() [source]

Get or compute the number of cols.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(1, 0, 2),
...                           MatrixEntry(2, 1, 3.7)])
```

```
>>> mat = CoordinateMatrix(entries)
>>> print(mat.numCols())
2
```

```
>>> mat = CoordinateMatrix(entries, 7, 6)
>>> print(mat.numCols())
6
```

numRows() [source]

Get or compute the number of rows.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(1, 0, 2),
...                           MatrixEntry(2, 1, 3.7)])
```

```
>>> mat = CoordinateMatrix(entries)
>>> print(mat.numRows())
3
```

```
>>> mat = CoordinateMatrix(entries, 7, 6)
>>> print(mat.numRows())
7
```

toBlockMatrix(*rowsPerBlock=1024*, *colsPerBlock=1024*) [source]

Convert this matrix to a BlockMatrix.

Parameters: • **rowsPerBlock** – Number of rows that make up each block. The blocks forming the final rows are not required to have the

given number of rows.
- **colsPerBlock** – Number of columns that make up each block. The blocks forming the final columns are not required to have the given number of columns.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(6, 4, 2.1)])
>>> mat = CoordinateMatrix(entries).toBlockMatrix()
```

```
>>> # This CoordinateMatrix will have 7 effective rows, due to
>>> # the highest row index being 6, and the ensuing
>>> # BlockMatrix will have 7 rows as well.
>>> print(mat.numRows())
7
```

```
>>> # This CoordinateMatrix will have 5 columns, due to the
>>> # highest column index being 4, and the ensuing
>>> # BlockMatrix will have 5 columns as well.
>>> print(mat.numCols())
5
```

toIndexedRowMatrix() [source]

Convert this matrix to an IndexedRowMatrix.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(6, 4, 2.1)])
>>> mat = CoordinateMatrix(entries).toIndexedRowMatrix()
```

```
>>> # This CoordinateMatrix will have 7 effective rows, due to
>>> # the highest row index being 6, and the ensuing
>>> # IndexedRowMatrix will have 7 rows as well.
>>> print(mat.numRows())
7
```

```
>>> # This CoordinateMatrix will have 5 columns, due to the
>>> # highest column index being 4, and the ensuing
```

```
>>> # IndexedRowMatrix will have 5 columns as well.
>>> print(mat.numCols())
5
```

toRowMatrix() [source]

Convert this matrix to a RowMatrix.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(6, 4, 2.1)])
>>> mat = CoordinateMatrix(entries).toRowMatrix()
```

```
>>> # This CoordinateMatrix will have 7 effective rows, due to
>>> # the highest row index being 6, but the ensuing RowMatrix
>>> # will only have 2 rows since there are only entries on 2
>>> # unique rows.
>>> print(mat.numRows())
2
```

```
>>> # This CoordinateMatrix will have 5 columns, due to the
>>> # highest column index being 4, and the ensuing RowMatrix
>>> # will have 5 columns as well.
>>> print(mat.numCols())
5
```

transpose() [source]

Transpose this CoordinateMatrix.

```
>>> entries = sc.parallelize([MatrixEntry(0, 0, 1.2),
...                           MatrixEntry(1, 0, 2),
...                           MatrixEntry(2, 1, 3.7)])
>>> mat = CoordinateMatrix(entries)
>>> mat_transposed = mat.transpose()
```

```
>>> print(mat_transposed.numRows())
2
```

```
>>> print(mat_transposed.numCols())
3
```

New in version 2.0.0.

class pyspark.mllib.linalg.distributed.**BlockMatrix**(*blocks*, *rowsPerBlock*, *colsPerBlock*, *numRows=0*, *numCols=0*) [source]

Bases: **pyspark.mllib.linalg.distributed.DistributedMatrix**

Represents a distributed matrix in blocks of local matrices.

Parameters:
- **blocks** – An RDD of sub-matrix blocks ((blockRowIndex, blockColIndex), sub-matrix) that form this distributed matrix. If multiple blocks with the same index exist, the results for operations like add and multiply will be unpredictable.
- **rowsPerBlock** – Number of rows that make up each block. The blocks forming the final rows are not required to have the given number of rows.
- **colsPerBlock** – Number of columns that make up each block. The blocks forming the final columns are not required to have the given number of columns.
- **numRows** – Number of rows of this matrix. If the supplied value is less than or equal to zero, the number of rows will be calculated when *numRows* is invoked.
- **numCols** – Number of columns of this matrix. If the supplied value is less than or equal to zero, the number of columns will be calculated when *numCols* is invoked.

add(*other*) [source]

Adds two block matrices together. The matrices must have the same size and matching *rowsPerBlock* and *colsPerBlock* values. If one of the sub matrix blocks that are being added is a SparseMatrix, the resulting sub matrix block will also be a SparseMatrix, even if it is being added to a DenseMatrix. If two dense sub matrix blocks are added, the output block will also be a DenseMatrix.

```
>>> dm1 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
>>> dm2 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [7, 11, 12])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm2)])
>>> mat1 = BlockMatrix(blocks1, 3, 2)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.add(mat2).toLocalMatrix()
DenseMatrix(6, 2, [2.0, 4.0, 6.0, 14.0, 16.0, 18.0, 8.0, 10.0, 12.0, 20.0, 22.0, 24.0], 0)
```

```
>>> mat1.add(mat3).toLocalMatrix()
DenseMatrix(6, 2, [8.0, 2.0, 3.0, 14.0, 16.0, 18.0, 4.0, 16.0, 18.0, 20.0, 22.0, 24.0], 0)
```

blocks [source]

The RDD of sub-matrix blocks ((blockRowIndex, blockColIndex), sub-matrix) that form this distributed matrix.

```
>>> mat = BlockMatrix(
...     sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                     ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))]), 3, 2)
>>> blocks = mat.blocks
>>> blocks.first()
((0, 0), DenseMatrix(3, 2, [1.0, 2.0, 3.0, 4.0, 5.0, 6.0], 0))
```

cache() [source]

Caches the underlying RDD.

New in version 2.0.0.

colsPerBlock [source]

Number of columns that make up each block.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2)
>>> mat.colsPerBlock
2
```

multiply(*other*) [source]

Left multiplies this BlockMatrix by *other*, another BlockMatrix. The *colsPerBlock* of this matrix must equal the *rowsPerBlock* of *other*. If *other* contains any SparseMatrix blocks, they will have to be converted to DenseMatrix blocks. The output BlockMatrix will only consist of DenseMatrix blocks. This may cause some performance issues until support for multiplying two sparse matrices is added.

```
>>> dm1 = Matrices.dense(2, 3, [1, 2, 3, 4, 5, 6])
>>> dm2 = Matrices.dense(2, 3, [7, 8, 9, 10, 11, 12])
>>> dm3 = Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])
>>> dm4 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [7, 11, 12])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((0, 1), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm3), ((1, 0), dm4)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm4)])
>>> mat1 = BlockMatrix(blocks1, 2, 3)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.multiply(mat2).toLocalMatrix()
DenseMatrix(2, 2, [242.0, 272.0, 350.0, 398.0], 0)
```

```
>>> mat1.multiply(mat3).toLocalMatrix()
DenseMatrix(2, 2, [227.0, 258.0, 394.0, 450.0], 0)
```

numColBlocks [source]

Number of columns of blocks in the BlockMatrix.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2)
>>> mat.numColBlocks
1
```

numCols() [source]

Get or compute the number of cols.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
```

```
>>> mat = BlockMatrix(blocks, 3, 2)
>>> print(mat.numCols())
```

```
2
```

```
>>> mat = BlockMatrix(blocks, 3, 2, 7, 6)
>>> print(mat.numCols())
6
```

numRowBlocks [source]

Number of rows of blocks in the BlockMatrix.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2)
>>> mat.numRowBlocks
2
```

numRows() [source]

Get or compute the number of rows.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
```

```
>>> mat = BlockMatrix(blocks, 3, 2)
>>> print(mat.numRows())
6
```

```
>>> mat = BlockMatrix(blocks, 3, 2, 7, 6)
>>> print(mat.numRows())
7
```

persist(*storageLevel*) [source]

Persists the underlying RDD with the specified storage level.

New in version 2.0.0.

rowsPerBlock [source]

Number of rows that make up each block.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2)
>>> mat.rowsPerBlock
3
```

subtract(*other*) [source]

Subtracts the given block matrix *other* from this block matrix: *this* - *other*. The matrices must have the same size and matching *rowsPerBlock* and *colsPerBlock* values. If one of the sub matrix blocks that are being subtracted is a SparseMatrix, the resulting sub matrix block will also be a SparseMatrix, even if it is being subtracted from a DenseMatrix. If two dense sub matrix blocks are subtracted, the output block will also be a DenseMatrix.

```
>>> dm1 = Matrices.dense(3, 2, [3, 1, 5, 4, 6, 2])
>>> dm2 = Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12])
>>> sm = Matrices.sparse(3, 2, [0, 1, 3], [0, 1, 2], [1, 2, 3])
>>> blocks1 = sc.parallelize([((0, 0), dm1), ((1, 0), dm2)])
>>> blocks2 = sc.parallelize([((0, 0), dm2), ((1, 0), dm1)])
>>> blocks3 = sc.parallelize([((0, 0), sm), ((1, 0), dm2)])
>>> mat1 = BlockMatrix(blocks1, 3, 2)
>>> mat2 = BlockMatrix(blocks2, 3, 2)
>>> mat3 = BlockMatrix(blocks3, 3, 2)
```

```
>>> mat1.subtract(mat2).toLocalMatrix()
DenseMatrix(6, 2, [-4.0, -7.0, -4.0, 4.0, 7.0, 4.0, -6.0, -5.0, -10.0, 6.0, 5.0, 10.0], 0)
```

```
>>> mat2.subtract(mat3).toLocalMatrix()
DenseMatrix(6, 2, [6.0, 8.0, 9.0, -4.0, -7.0, -4.0, 10.0, 9.0, 9.0, -6.0, -5.0, -10.0], 0)
```

New in version 2.0.0.

toCoordinateMatrix() [source]

Convert this matrix to a CoordinateMatrix.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(1, 2, [1, 2])),
...                          ((1, 0), Matrices.dense(1, 2, [7, 8]))])
>>> mat = BlockMatrix(blocks, 1, 2).toCoordinateMatrix()
>>> mat.entries.take(3)
[MatrixEntry(0, 0, 1.0), MatrixEntry(0, 1, 2.0), MatrixEntry(1, 0, 7.0)]
```

toIndexedRowMatrix() [source]

Convert this matrix to an IndexedRowMatrix.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2).toIndexedRowMatrix()
```

```
>>> # This BlockMatrix will have 6 effective rows, due to
>>> # having two sub-matrix blocks stacked, each with 3 rows.
>>> # The ensuing IndexedRowMatrix will also have 6 rows.
>>> print(mat.numRows())
6
```

```
>>> # This BlockMatrix will have 2 effective columns, due to
>>> # having two sub-matrix blocks stacked, each with 2 columns.
>>> # The ensuing IndexedRowMatrix will also have 2 columns.
>>> print(mat.numCols())
2
```

toLocalMatrix() [source]

Collect the distributed matrix on the driver as a DenseMatrix.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2).toLocalMatrix()
```

```
>>> # This BlockMatrix will have 6 effective rows, due to
>>> # having two sub-matrix blocks stacked, each with 3 rows.
>>> # The ensuing DenseMatrix will also have 6 rows.
```

```
>>> print(mat.numRows)
6
```

```
>>> # This BlockMatrix will have 2 effective columns, due to
>>> # having two sub-matrix blocks stacked, each with 2
>>> # columns. The ensuing DenseMatrix will also have 2 columns.
>>> print(mat.numCols)
2
```

transpose() [source]

Transpose this BlockMatrix. Returns a new BlockMatrix instance sharing the same underlying data. Is a lazy operation.

```
>>> blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
...                          ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
>>> mat = BlockMatrix(blocks, 3, 2)
```

```
>>> mat_transposed = mat.transpose()
>>> mat_transposed.toLocalMatrix()
DenseMatrix(2, 6, [1.0, 4.0, 2.0, 5.0, 3.0, 6.0, 7.0, 10.0, 8.0, 11.0, 9.0, 12.0], 0)
```

New in version 2.0.0.

validate() [source]

Validates the block matrix info against the matrix data (*blocks*) and throws an exception if any error is found.

New in version 2.0.0.

pyspark.mllib.random module

Python package for random data generation.

class pyspark.mllib.random.**RandomRDDs** [source]

Generator methods for creating RDDs comprised of i.i.d samples from some distribution.

New in version 1.1.0.

static **exponentialRDD**(*sc*, *mean*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the Exponential distribution with the input mean.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – Mean, or 1 / lambda, for the Exponential distribution.
- **size** – Size of the RDD.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ Exp(mean).

```
>>> mean = 2.0
>>> x = RandomRDDs.exponentialRDD(sc, mean, 1000, seed=2)
>>> stats = x.stats()
>>> stats.count()
1000
>>> abs(stats.mean() - mean) < 0.5
True
>>> from math import sqrt
>>> abs(stats.stdev() - sqrt(mean)) < 0.5
True
```

New in version 1.3.0.

static **exponentialVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the Exponential distribution with the input mean.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – Mean, or 1 / lambda, for the Exponential distribution.
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*)
- **seed** – Random seed (default: a random long integer).

Returns: RDD of Vector with vectors containing i.i.d. samples ~ Exp(mean).

```
>>> import numpy as np
>>> mean = 0.5
>>> rdd = RandomRDDs.exponentialVectorRDD(sc, mean, 100, 100, seed=1)
>>> mat = np.mat(rdd.collect())
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - mean) < 0.5
True
>>> from math import sqrt
>>> abs(mat.std() - sqrt(mean)) < 0.5
True
```

New in version 1.3.0.

static **gammaRDD**(*sc*, *shape*, *scale*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the Gamma distribution with the input shape and scale.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **shape** – shape (> 0) parameter for the Gamma distribution
- **scale** – scale (> 0) parameter for the Gamma distribution
- **size** – Size of the RDD.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ Gamma(shape, scale).

```
>>> from math import sqrt
>>> shape = 1.0
>>> scale = 2.0
>>> expMean = shape * scale
>>> expStd = sqrt(shape * scale * scale)
>>> x = RandomRDDs.gammaRDD(sc, shape, scale, 1000, seed=2)
>>> stats = x.stats()
>>> stats.count()
1000
>>> abs(stats.mean() - expMean) < 0.5
True
>>> abs(stats.stdev() - expStd) < 0.5
True
```

New in version 1.3.0.

static **gammaVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the Gamma distribution.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **shape** – Shape (> 0) of the Gamma distribution
- **scale** – Scale (> 0) of the Gamma distribution
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of Vector with vectors containing i.i.d. samples ~ Gamma(shape, scale).

```
>>> import numpy as np
>>> from math import sqrt
>>> shape = 1.0
>>> scale = 2.0
>>> expMean = shape * scale
>>> expStd = sqrt(shape * scale * scale)
>>> mat = np.matrix(RandomRDDs.gammaVectorRDD(sc, shape, scale, 100, 100, seed=1).collect())
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - expMean) < 0.1
True
>>> abs(mat.std() - expStd) < 0.1
True
```

New in version 1.3.0.

static **logNormalRDD**(*sc*, *mean*, *std*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the log normal distribution with the input mean and standard distribution.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – mean for the log Normal distribution
- **std** – std for the log Normal distribution
- **size** – Size of the RDD.

- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ log N(mean, std).

```
>>> from math import sqrt, exp
>>> mean = 0.0
>>> std = 1.0
>>> expMean = exp(mean + 0.5 * std * std)
>>> expStd = sqrt((exp(std * std) - 1.0) * exp(2.0 * mean + std * std))
>>> x = RandomRDDs.logNormalRDD(sc, mean, std, 1000, seed=2)
>>> stats = x.stats()
>>> stats.count()
1000
>>> abs(stats.mean() - expMean) < 0.5
True
>>> from math import sqrt
>>> abs(stats.stdev() - expStd) < 0.5
True
```

New in version 1.3.0.

static **logNormalVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the log normal distribution.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – Mean of the log normal distribution
- **std** – Standard Deviation of the log normal distribution
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of Vector with vectors containing i.i.d. samples ~ log *N(mean, std)*.

```
>>> import numpy as np
>>> from math import sqrt, exp
>>> mean = 0.0
>>> std = 1.0
>>> expMean = exp(mean + 0.5 * std * std)
```

```
>>> expStd = sqrt((exp(std * std) - 1.0) * exp(2.0 * mean + std * std))
>>> m = RandomRDDs.logNormalVectorRDD(sc, mean, std, 100, 100, seed=1).collect()
>>> mat = np.matrix(m)
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - expMean) < 0.1
True
>>> abs(mat.std() - expStd) < 0.1
True
```

New in version 1.3.0.

static **normalRDD**(*sc*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the standard normal distribution.

To transform the distribution in the generated RDD from standard normal to some other normal N(mean, sigma^2), use *RandomRDDs.normal(sc, n, p, seed) .map(lambda v: mean + sigma * v)*

Parameters:
- **sc** – SparkContext used to create the RDD.
- **size** – Size of the RDD.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ N(0.0, 1.0).

```
>>> x = RandomRDDs.normalRDD(sc, 1000, seed=1)
>>> stats = x.stats()
>>> stats.count()
1000
>>> abs(stats.mean() - 0.0) < 0.1
True
>>> abs(stats.stdev() - 1.0) < 0.1
True
```

New in version 1.1.0.

static **normalVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the standard normal distribution.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of Vector with vectors containing i.i.d. samples ~ *N(0.0, 1.0)*.

```
>>> import numpy as np
>>> mat = np.matrix(RandomRDDs.normalVectorRDD(sc, 100, 100, seed=1).collect())
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - 0.0) < 0.1
True
>>> abs(mat.std() - 1.0) < 0.1
True
```

New in version 1.1.0.

static **poissonRDD**(*sc*, *mean*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the Poisson distribution with the input mean.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – Mean, or lambda, for the Poisson distribution.
- **size** – Size of the RDD.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ Pois(mean).

```
>>> mean = 100.0
>>> x = RandomRDDs.poissonRDD(sc, mean, 1000, seed=2)
>>> stats = x.stats()
>>> stats.count()
1000
>>> abs(stats.mean() - mean) < 0.5
True
>>> from math import sqrt
>>> abs(stats.stdev() - sqrt(mean)) < 0.5
True
```

New in version 1.1.0.

static **poissonVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the Poisson distribution with the input mean.

Parameters:
- **sc** – SparkContext used to create the RDD.
- **mean** – Mean, or lambda, for the Poisson distribution.
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*)
- **seed** – Random seed (default: a random long integer).

Returns: RDD of Vector with vectors containing i.i.d. samples ~ Pois(mean).

```
>>> import numpy as np
>>> mean = 100.0
>>> rdd = RandomRDDs.poissonVectorRDD(sc, mean, 100, 100, seed=1)
>>> mat = np.mat(rdd.collect())
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - mean) < 0.5
True
>>> from math import sqrt
>>> abs(mat.std() - sqrt(mean)) < 0.5
True
```

New in version 1.1.0.

static **uniformRDD**(*sc*, *size*, *numPartitions=None*, *seed=None*) [source]

Generates an RDD comprised of i.i.d. samples from the uniform distribution U(0.0, 1.0).

To transform the distribution in the generated RDD from U(0.0, 1.0) to U(a, b), use *RandomRDDs.uniformRDD(sc, n, p, seed) .map(lambda v: a + (b - a) * v)*

Parameters:
- **sc** – SparkContext used to create the RDD.
- **size** – Size of the RDD.

- **numPartitions** – Number of partitions in the RDD (default: *sc.defaultParallelism*).
- **seed** – Random seed (default: a random long integer).

Returns: RDD of float comprised of i.i.d. samples ~ *U(0.0, 1.0)*.

```
>>> x = RandomRDDs.uniformRDD(sc, 100).collect()
>>> len(x)
100
>>> max(x) <= 1.0 and min(x) >= 0.0
True
>>> RandomRDDs.uniformRDD(sc, 100, 4).getNumPartitions()
4
>>> parts = RandomRDDs.uniformRDD(sc, 100, seed=4).getNumPartitions()
>>> parts == sc.defaultParallelism
True
```

New in version 1.1.0.

static **uniformVectorRDD**(*sc*, **a*, ***kw*) [source]

Generates an RDD comprised of vectors containing i.i.d. samples drawn from the uniform distribution U(0.0, 1.0).

Parameters:
- **sc** – SparkContext used to create the RDD.
- **numRows** – Number of Vectors in the RDD.
- **numCols** – Number of elements in each Vector.
- **numPartitions** – Number of partitions in the RDD.
- **seed** – Seed for the RNG that generates the seed for the generator in each partition.

Returns: RDD of Vector with vectors containing i.i.d samples ~ *U(0.0, 1.0)*.

```
>>> import numpy as np
>>> mat = np.matrix(RandomRDDs.uniformVectorRDD(sc, 10, 10).collect())
>>> mat.shape
(10, 10)
>>> mat.max() <= 1.0 and mat.min() >= 0.0
True
>>> RandomRDDs.uniformVectorRDD(sc, 10, 10, 4).getNumPartitions()
4
```

New in version 1.1.0.

pyspark.mllib.recommendation module

class pyspark.mllib.recommendation.**MatrixFactorizationModel**(*java_model*) [source]

A matrix factorisation model trained by regularized alternating least-squares.

```
>>> r1 = (1, 1, 1.0)
>>> r2 = (1, 2, 2.0)
>>> r3 = (2, 1, 2.0)
>>> ratings = sc.parallelize([r1, r2, r3])
>>> model = ALS.trainImplicit(ratings, 1, seed=10)
>>> model.predict(2, 2)
0.4...
```

```
>>> testset = sc.parallelize([(1, 2), (1, 1)])
>>> model = ALS.train(ratings, 2, seed=0)
>>> model.predictAll(testset).collect()
[Rating(user=1, product=1, rating=1.0...), Rating(user=1, product=2, rating=1.9...)]
```

```
>>> model = ALS.train(ratings, 4, seed=10)
>>> model.userFeatures().collect()
[(1, array('d', [...])), (2, array('d', [...]))]
```

```
>>> model.recommendUsers(1, 2)
[Rating(user=2, product=1, rating=1.9...), Rating(user=1, product=1, rating=1.0...)]
>>> model.recommendProducts(1, 2)
[Rating(user=1, product=2, rating=1.9...), Rating(user=1, product=1, rating=1.0...)]
>>> model.rank
4
```

```
>>> first_user = model.userFeatures().take(1)[0]
>>> latents = first_user[1]
>>> len(latents)
4
```

```
>>> model.productFeatures().collect()
```

```
[(1, array('d', [...])), (2, array('d', [...]))]
```

```
>>> first_product = model.productFeatures().take(1)[0]
>>> latents = first_product[1]
>>> len(latents)
4
```

```
>>> products_for_users = model.recommendProductsForUsers(1).collect()
>>> len(products_for_users)
2
>>> products_for_users[0]
(1, (Rating(user=1, product=2, rating=...),))
```

```
>>> users_for_products = model.recommendUsersForProducts(1).collect()
>>> len(users_for_products)
2
>>> users_for_products[0]
(1, (Rating(user=2, product=1, rating=...),))
```

```
>>> model = ALS.train(ratings, 1, nonnegative=True, seed=10)
>>> model.predict(2, 2)
3.73...
```

```
>>> df = sqlContext.createDataFrame([Rating(1, 1, 1.0), Rating(1, 2, 2.0), Rating(2, 1, 2.0)])
>>> model = ALS.train(df, 1, nonnegative=True, seed=10)
>>> model.predict(2, 2)
3.73...
```

```
>>> model = ALS.trainImplicit(ratings, 1, nonnegative=True, seed=10)
>>> model.predict(2, 2)
0.4...
```

```
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> model.save(sc, path)
```

```
>>> sameModel = MatrixFactorizationModel.load(sc, path)
>>> sameModel.predict(2, 2)
0.4...
>>> sameModel.predictAll(testset).collect()
[Rating(...
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 0.9.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path

New in version 1.3.1.

predict(*user*, *product*) [source]

Predicts rating for the given user and product.

New in version 0.9.0.

predictAll(*user_product*) [source]

Returns a list of predicted ratings for input user and product pairs.

New in version 0.9.0.

productFeatures() [source]

Returns a paired RDD, where the first element is the product and the second is an array of features corresponding to that product.

New in version 1.2.0.

rank [source]

Rank for the features in this model

New in version 1.4.0.

recommendProducts(*user*, *num*) [source]

Recommends the top “num” number of products for a given user and returns a list of Rating objects sorted by the predicted rating in descending order.

New in version 1.4.0.

recommendProductsForUsers(*num*) [source]

Recommends the top “num” number of products for all users. The number of recommendations returned per user may be less than “num”.

recommendUsers(*product*, *num*) [source]

Recommends the top “num” number of users for a given product and returns a list of Rating objects sorted by the predicted rating in descending order.

New in version 1.4.0.

recommendUsersForProducts(*num*) [source]

Recommends the top “num” number of users for all products. The number of recommendations returned per product may be less than “num”.

userFeatures() [source]

Returns a paired RDD, where the first element is the user and the second is an array of features corresponding to that user.

New in version 1.2.0.

class pyspark.mllib.recommendation.**ALS** [source]

Alternating Least Squares matrix factorization

New in version 0.9.0.

classmethod **train**(*ratings*, *rank*, *iterations=5*, *lambda_=0.01*, *blocks=-1*, *nonnegative=False*, *seed=None*) [source]

Train a matrix factorization model given an RDD of ratings by users for a subset of products. The ratings matrix is approximated as the product of two lower-rank matrices of a given rank (number of features). To solve for these features, ALS is run iteratively with a configurable level of parallelism.

Parameters:
- **ratings** – RDD of *Rating* or (userID, productID, rating) tuple.
- **rank** – Rank of the feature matrices computed (number of features).
- **iterations** – Number of iterations of ALS. (default: 5)

- **lambda** – Regularization parameter. (default: 0.01)
- **blocks** – Number of blocks used to parallelize the computation. A value of -1 will use an auto-configured number of blocks. (default: -1)
- **nonnegative** – A value of True will solve least-squares with nonnegativity constraints. (default: False)
- **seed** – Random seed for initial matrix factorization model. A value of None will use system time as the seed. (default: None)

New in version 0.9.0.

classmethod **trainImplicit**(*ratings*, *rank*, *iterations=5*, *lambda_=0.01*, *blocks=-1*, *alpha=0.01*, *nonnegative=False*, *seed=None*) [source]

Train a matrix factorization model given an RDD of 'implicit preferences' of users for a subset of products. The ratings matrix is approximated as the product of two lower-rank matrices of a given rank (number of features). To solve for these features, ALS is run iteratively with a configurable level of parallelism.

Parameters:
- **ratings** – RDD of *Rating* or (userID, productID, rating) tuple.
- **rank** – Rank of the feature matrices computed (number of features).
- **iterations** – Number of iterations of ALS. (default: 5)
- **lambda** – Regularization parameter. (default: 0.01)
- **blocks** – Number of blocks used to parallelize the computation. A value of -1 will use an auto-configured number of blocks. (default: -1)
- **alpha** – A constant used in computing confidence. (default: 0.01)
- **nonnegative** – A value of True will solve least-squares with nonnegativity constraints. (default: False)
- **seed** – Random seed for initial matrix factorization model. A value of None will use system time as the seed. (default: None)

New in version 0.9.0.

class pyspark.mllib.recommendation.**Rating** [source]

Represents a (user, product, rating) tuple.

```
>>> r = Rating(1, 2, 5.0)
>>> (r.user, r.product, r.rating)
(1, 2, 5.0)
>>> (r[0], r[1], r[2])
(1, 2, 5.0)
```

New in version 1.2.0.

pyspark.mllib.regression module

class pyspark.mllib.regression.**LabeledPoint**(*label*, *features*) [source]

Class that represents the features and labels of a data point.

Parameters:
- **label** – Label for this data point.
- **features** – Vector of features for this point (NumPy array, list, pyspark.mllib.linalg.SparseVector, or scipy.sparse column matrix).

Note: ‘label’ and ‘features’ are accessible as class attributes.

New in version 1.0.0.

class pyspark.mllib.regression.**LinearModel**(*weights*, *intercept*) [source]

A linear model that has a vector of coefficients and an intercept.

Parameters:
- **weights** – Weights computed for every feature.
- **intercept** – Intercept computed for this model.

New in version 0.9.0.

intercept [source]

Intercept computed for this model.

New in version 1.0.0.

weights [source]

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.regression.**LinearRegressionModel**(*weights*, *intercept*) [source]

A linear regression model derived from a least-squares fit.

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(1.0, [1.0]),
```

```
...     LabeledPoint(3.0, [2.0]),
...     LabeledPoint(2.0, [3.0])
... ]
>>> lrm = LinearRegressionWithSGD.train(sc.parallelize(data), iterations=10,
...     initialWeights=np.array([1.0]))
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> abs(lrm.predict(sc.parallelize([[1.0]])).collect()[0] - 1) < 0.5
True
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> lrm.save(sc, path)
>>> sameModel = LinearRegressionModel.load(sc, path)
>>> abs(sameModel.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(sameModel.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(sameModel.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except:
...     pass
>>> data = [
...     LabeledPoint(0.0, SparseVector(1, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(1, {0: 1.0})),
...     LabeledPoint(3.0, SparseVector(1, {0: 2.0})),
...     LabeledPoint(2.0, SparseVector(1, {0: 3.0}))
... ]
>>> lrm = LinearRegressionWithSGD.train(sc.parallelize(data), iterations=10,
...     initialWeights=array([1.0]))
>>> abs(lrm.predict(array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> lrm = LinearRegressionWithSGD.train(sc.parallelize(data), iterations=10, step=1.0,
...    miniBatchFraction=1.0, initialWeights=array([1.0]), regParam=0.1, regType="l2",
...    intercept=True, validateData=True)
>>> abs(lrm.predict(array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
```

New in version 0.9.0.

intercept

Intercept computed for this model.

New in version 1.0.0.

classmethod **load**(*sc*, *path*) [source]

Load a LinearRegressionModel.

New in version 1.4.0.

predict(*x*)

Predict the value of the dependent variable given a vector or an RDD of vectors containing values for the independent variables.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save a LinearRegressionModel.

New in version 1.4.0.

weights

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.regression.**LinearRegressionWithSGD** [source]

New in version 0.9.0.

> **Note:** Deprecated in 2.0.0. Use ml.regression.LinearRegression.

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *miniBatchFraction=1.0*, *initialWeights=None*, *regParam=0.0*, *regType=None*, *intercept=False*, *validateData=True*, *convergenceTol=0.001*) [source]

Train a linear regression model using Stochastic Gradient Descent (SGD). This solves the least squares regression formulation

f(weights) = 1/(2n) ||A weights - y||^2

which is the mean squared error. Here the data matrix has n rows, and the input RDD holds the set of rows of A, each with its corresponding right hand side label y. See also the documentation for the precise formulation.

Parameters:

- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **step** – The step parameter used in SGD. (default: 1.0)
- **miniBatchFraction** – Fraction of data to be used for each SGD iteration. (default: 1.0)
- **initialWeights** – The initial weights. (default: None)
- **regParam** – The regularizer parameter. (default: 0.0)
- **regType** –
  The type of regularizer used for training our model. Supported values:
  - “l1” for using L1 regularization
  - “l2” for using L2 regularization
  - None for no regularization (default)
- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e., whether bias features are activated or not). (default: False)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **convergenceTol** – A condition which decides iteration termination. (default: 0.001)

New in version 0.9.0.

class pyspark.mllib.regression.**RidgeRegressionModel**(*weights*, *intercept*) [source]

A linear regression model derived from a least-squares fit with an l_2 penalty term.

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(1.0, [1.0]),
...     LabeledPoint(3.0, [2.0]),
...     LabeledPoint(2.0, [3.0])
... ]
>>> lrm = RidgeRegressionWithSGD.train(sc.parallelize(data), iterations=10,
...     initialWeights=array([1.0]))
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
```

```
True
>>> abs(lrm.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> abs(lrm.predict(sc.parallelize([[1.0]])).collect()[0] - 1) < 0.5
True
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> lrm.save(sc, path)
>>> sameModel = RidgeRegressionModel.load(sc, path)
>>> abs(sameModel.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(sameModel.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(sameModel.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> from shutil import rmtree
>>> try:
...    rmtree(path)
... except:
...    pass
>>> data = [
...     LabeledPoint(0.0, SparseVector(1, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(1, {0: 1.0})),
...     LabeledPoint(3.0, SparseVector(1, {0: 2.0})),
...     LabeledPoint(2.0, SparseVector(1, {0: 3.0}))
... ]
>>> lrm = LinearRegressionWithSGD.train(sc.parallelize(data), iterations=10,
...     initialWeights=array([1.0]))
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> lrm = RidgeRegressionWithSGD.train(sc.parallelize(data), iterations=10, step=1.0,
...     regParam=0.01, miniBatchFraction=1.0, initialWeights=array([1.0]), intercept=True,
...     validateData=True)
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
```

New in version 0.9.0.

intercept

Intercept computed for this model.

New in version 1.0.0.

classmethod **load**(*sc*, *path*) [source]

Load a RidgeRegressionMode.

New in version 1.4.0.

predict(*x*)

Predict the value of the dependent variable given a vector or an RDD of vectors containing values for the independent variables.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save a RidgeRegressionMode.

New in version 1.4.0.

weights

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.regression.**RidgeRegressionWithSGD** [source]

New in version 0.9.0.

> **Note:** Deprecated in 2.0.0. Use ml.regression.LinearRegression with elasticNetParam = 0.0. Note the default regParam is 0.01 for RidgeRegressionWithSGD, but is 0.0 for LinearRegression.

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *regParam=0.01*, *miniBatchFraction=1.0*, *initialWeights=None*, *intercept=False*, *validateData=True*, *convergenceTol=0.001*) [source]

Train a regression model with L2-regularization using Stochastic Gradient Descent. This solves the l2-regularized least squares regression formulation

f(weights) = 1/(2n) ||A weights - y||^2 + regParam/2 ||weights||^2

Here the data matrix has n rows, and the input RDD holds the set of rows of A, each with its corresponding right hand side label y. See also the documentation for the precise formulation.

Parameters:
- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **step** – The step parameter used in SGD. (default: 1.0)
- **regParam** – The regularizer parameter. (default: 0.01)
- **miniBatchFraction** – Fraction of data to be used for each SGD iteration. (default: 1.0)
- **initialWeights** – The initial weights. (default: None)
- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not). (default: False)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **convergenceTol** – A condition which decides iteration termination. (default: 0.001)

New in version 0.9.0.

class pyspark.mllib.regression.**LassoModel**(*weights*, *intercept*) [source]

A linear regression model derived from a least-squares fit with an l_1 penalty term.

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(1.0, [1.0]),
...     LabeledPoint(3.0, [2.0]),
...     LabeledPoint(2.0, [3.0])
... ]
>>> lrm = LassoWithSGD.train(sc.parallelize(data), iterations=10, initialWeights=array([1.0]))
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> abs(lrm.predict(sc.parallelize([[1.0]])).collect()[0] - 1) < 0.5
True
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> lrm.save(sc, path)
>>> sameModel = LassoModel.load(sc, path)
>>> abs(sameModel.predict(np.array([0.0])) - 0) < 0.5
```

```
True
>>> abs(sameModel.predict(np.array([1.0])) - 1) < 0.5
True
>>> abs(sameModel.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> from shutil import rmtree
>>> try:
...    rmtree(path)
... except:
...    pass
>>> data = [
...     LabeledPoint(0.0, SparseVector(1, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(1, {0: 1.0})),
...     LabeledPoint(3.0, SparseVector(1, {0: 2.0})),
...     LabeledPoint(2.0, SparseVector(1, {0: 3.0}))
... ]
>>> lrm = LinearRegressionWithSGD.train(sc.parallelize(data), iterations=10,
...     initialWeights=array([1.0]))
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
>>> lrm = LassoWithSGD.train(sc.parallelize(data), iterations=10, step=1.0,
...     regParam=0.01, miniBatchFraction=1.0, initialWeights=array([1.0]), intercept=True,
...     validateData=True)
>>> abs(lrm.predict(np.array([0.0])) - 0) < 0.5
True
>>> abs(lrm.predict(SparseVector(1, {0: 1.0})) - 1) < 0.5
True
```

New in version 0.9.0.

intercept

Intercept computed for this model.

New in version 1.0.0.

classmethod **load**(*sc*, *path*) [source]

Load a LassoModel.

New in version 1.4.0.

predict(*x*)

Predict the value of the dependent variable given a vector or an RDD of vectors containing values for the independent variables.

New in version 0.9.0.

save(*sc*, *path*) [source]

Save a LassoModel.

New in version 1.4.0.

weights

Weights computed for every feature.

New in version 1.0.0.

class pyspark.mllib.regression.**LassoWithSGD** [source]

New in version 0.9.0.

> **Note:** Deprecated in 2.0.0. Use ml.regression.LinearRegression with elasticNetParam = 1.0. Note the default regParam is 0.01 for LassoWithSGD, but is 0.0 for LinearRegression.

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *regParam=0.01*, *miniBatchFraction=1.0*, *initialWeights=None*, *intercept=False*, *validateData=True*, *convergenceTol=0.001*) [source]

Train a regression model with L1-regularization using Stochastic Gradient Descent. This solves the l1-regularized least squares regression formulation

f(weights) = 1/(2n) ||A weights - y||^2 + regParam ||weights||_1

Here the data matrix has n rows, and the input RDD holds the set of rows of A, each with its corresponding right hand side label y. See also the documentation for the precise formulation.

Parameters:
- **data** – The training data, an RDD of LabeledPoint.
- **iterations** – The number of iterations. (default: 100)
- **step** – The step parameter used in SGD. (default: 1.0)
- **regParam** – The regularizer parameter. (default: 0.01)
- **miniBatchFraction** – Fraction of data to be used for each SGD iteration. (default: 1.0)
- **initialWeights** – The initial weights. (default: None)

- **intercept** – Boolean parameter which indicates the use or not of the augmented representation for training data (i.e. whether bias features are activated or not). (default: False)
- **validateData** – Boolean parameter which indicates if the algorithm should validate data before training. (default: True)
- **convergenceTol** – A condition which decides iteration termination. (default: 0.001)

New in version 0.9.0.

class pyspark.mllib.regression.**IsotonicRegressionModel**(*boundaries*, *predictions*, *isotonic*) [source]

Regression model for isotonic regression.

Parameters:
- **boundaries** – Array of boundaries for which predictions are known. Boundaries must be sorted in increasing order.
- **predictions** – Array of predictions associated to the boundaries at the same index. Results of isotonic regression and therefore monotone.
- **isotonic** – Indicates whether this is isotonic or antitonic.

```
>>> data = [(1, 0, 1), (2, 1, 1), (3, 2, 1), (1, 3, 1), (6, 4, 1), (17, 5, 1), (16, 6, 1)]
>>> irm = IsotonicRegression.train(sc.parallelize(data))
>>> irm.predict(3)
2.0
>>> irm.predict(5)
16.5
>>> irm.predict(sc.parallelize([3, 5])).collect()
[2.0, 16.5]
>>> import os, tempfile
>>> path = tempfile.mkdtemp()
>>> irm.save(sc, path)
>>> sameModel = IsotonicRegressionModel.load(sc, path)
>>> sameModel.predict(3)
2.0
>>> sameModel.predict(5)
16.5
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

New in version 1.4.0.

classmethod **load**(*sc*, *path*) [source]

Load an IsotonicRegressionModel.

New in version 1.4.0.

predict(*x*) [source]

Predict labels for provided features. Using a piecewise linear function. 1) If x exactly matches a boundary then associated prediction is returned. In case there are multiple predictions with the same boundary then one of them is returned. Which one is undefined (same as java.util.Arrays.binarySearch). 2) If x is lower or higher than all boundaries then first or last prediction is returned respectively. In case there are multiple predictions with the same boundary then the lowest or highest is returned respectively. 3) If x falls between two values in boundary array then prediction is treated as piecewise linear function and interpolated value is returned. In case there are multiple values with the same boundary then the same rules as in 2) are used.

Parameters: **x** – Feature or RDD of Features to be labeled.

New in version 1.4.0.

save(*sc*, *path*) [source]

Save an IsotonicRegressionModel.

New in version 1.4.0.

class pyspark.mllib.regression.**IsotonicRegression** [source]

Isotonic regression. Currently implemented using parallelized pool adjacent violators algorithm. Only univariate (single feature) algorithm supported.

Sequential PAV implementation based on:

> Tibshirani, Ryan J., Holger Hoefling, and Robert Tibshirani. “Nearly-isotonic regression.” Technometrics 53.1 (2011): 54-61. Available from http://www.stat.cmu.edu/~ryantibs/papers/neariso.pdf

Sequential PAV parallelization based on:

> Kearsley, Anthony J., Richard A. Tapia, and Michael W. Trosset. “An approach to parallelizing isotonic regression.” Applied Mathematics and Parallel Computing. Physica-Verlag HD, 1996. 141-147. Available from http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR96640.pdf

See Isotonic regression (Wikipedia).

New in version 1.4.0.

classmethod **train**(*data*, *isotonic=True*) [source]

Train an isotonic regression model on the given data.

Parameters:
- **data** – RDD of (label, feature, weight) tuples.
- **isotonic** – Whether this is isotonic (which is default) or antitonic. (default: True)

New in version 1.4.0.

class pyspark.mllib.regression.**StreamingLinearAlgorithm**(*model*) [source]

Base class that has to be inherited by any StreamingLinearAlgorithm.

Prevents reimplementation of methods predictOn and predictOnValues.

New in version 1.5.0.

latestModel() [source]

Returns the latest model.

New in version 1.5.0.

predictOn(*dstream*) [source]

Use the model to make predictions on batches of data from a DStream.

Returns: DStream containing predictions.

New in version 1.5.0.

predictOnValues(*dstream*) [source]

Use the model to make predictions on the values of a DStream and carry over its keys.

Returns: DStream containing the input keys and the predictions as values.

New in version 1.5.0.

class pyspark.mllib.regression.**StreamingLinearRegressionWithSGD**(*stepSize=0.1*, *numIterations=50*, *miniBatchFraction=1.0*, *convergenceTol=0.001*) [source]

Train or predict a linear regression model on streaming data. Training uses Stochastic Gradient Descent to update the model based on each new batch of incoming data from a DStream (see *LinearRegressionWithSGD* for model equation).

Each batch of data is assumed to be an RDD of LabeledPoints. The number of data points per batch can vary, but the number of features must be constant. An initial weight vector must be provided.

Parameters:
- **stepSize** – Step size for each iteration of gradient descent. (default: 0.1)
- **numIterations** – Number of iterations run for each batch of data. (default: 50)
- **miniBatchFraction** – Fraction of each batch of data to use for updates. (default: 1.0)
- **convergenceTol** – Value used to determine when to terminate iterations. (default: 0.001)

New in version 1.5.0.

`latestModel`()

Returns the latest model.

New in version 1.5.0.

`predictOn`(*dstream*)

Use the model to make predictions on batches of data from a DStream.

Returns: DStream containing predictions.

New in version 1.5.0.

`predictOnValues`(*dstream*)

Use the model to make predictions on the values of a DStream and carry over its keys.

Returns: DStream containing the input keys and the predictions as values.

New in version 1.5.0.

`setInitialWeights`(*initialWeights*) [source]

Set the initial value of weights.

This must be set before running trainOn and predictOn

New in version 1.5.0.

trainOn(*dstream*) [source]

Train the model on the incoming dstream.

New in version 1.5.0.

pyspark.mllib.stat module

Python package for statistical functions in MLlib.

class pyspark.mllib.stat.**Statistics**

static **chiSqTest**(*observed*, *expected=None*)

If *observed* is Vector, conduct Pearson's chi-squared goodness of fit test of the observed data against the expected distribution, or againt the uniform distribution (by default), with each category having an expected frequency of *1 / len(observed)*. (Note: *observed* cannot contain negative values)

If *observed* is matrix, conduct Pearson's independence test on the input contingency matrix, which cannot contain negative entries or columns or rows that sum up to 0.

If *observed* is an RDD of LabeledPoint, conduct Pearson's independence test for every feature against the label across the input RDD. For each feature, the (feature, label) pairs are converted into a contingency matrix for which the chi-squared statistic is computed. All label and feature values must be categorical.

Parameters:
- **observed** – it could be a vector containing the observed categorical counts/relative frequencies, or the contingency matrix (containing either counts or relative frequencies), or an RDD of LabeledPoint containing the labeled dataset with categorical features. Real-valued features will be treated as categorical for each distinct value.
- **expected** – Vector containing the expected categorical counts/relative frequencies. *expected* is rescaled if the *expected* sum differs from the *observed* sum.

Returns: ChiSquaredTest object containing the test statistic, degrees of freedom, p-value, the method used, and the null hypothesis.

```
>>> from pyspark.mllib.linalg import Vectors, Matrices
>>> observed = Vectors.dense([4, 6, 5])
>>> pearson = Statistics.chiSqTest(observed)
```

```
>>> print(pearson.statistic)
0.4
>>> pearson.degreesOfFreedom
2
>>> print(round(pearson.pValue, 4))
0.8187
>>> pearson.method
u'pearson'
>>> pearson.nullHypothesis
u'observed follows the same distribution as expected.'
```

```
>>> observed = Vectors.dense([21, 38, 43, 80])
>>> expected = Vectors.dense([3, 5, 7, 20])
>>> pearson = Statistics.chiSqTest(observed, expected)
>>> print(round(pearson.pValue, 4))
0.0027
```

```
>>> data = [40.0, 24.0, 29.0, 56.0, 32.0, 42.0, 31.0, 10.0, 0.0, 30.0, 15.0, 12.0]
>>> chi = Statistics.chiSqTest(Matrices.dense(3, 4, data))
>>> print(round(chi.statistic, 4))
21.9958
```

```
>>> data = [LabeledPoint(0.0, Vectors.dense([0.5, 10.0])),
...         LabeledPoint(0.0, Vectors.dense([1.5, 20.0])),
...         LabeledPoint(1.0, Vectors.dense([1.5, 30.0])),
...         LabeledPoint(0.0, Vectors.dense([3.5, 30.0])),
...         LabeledPoint(0.0, Vectors.dense([3.5, 40.0])),
...         LabeledPoint(1.0, Vectors.dense([3.5, 40.0])),]
>>> rdd = sc.parallelize(data, 4)
>>> chi = Statistics.chiSqTest(rdd)
>>> print(chi[0].statistic)
0.75
>>> print(chi[1].statistic)
1.5
```

static **colStats**(*rdd*)

Computes column-wise summary statistics for the input RDD[Vector].

Parameters: **rdd** – an RDD[Vector] for which column-wise summary statistics are to be computed.

Returns: `MultivariateStatisticalSummary` object containing column-wise summary statistics.

```
>>> from pyspark.mllib.linalg import Vectors
>>> rdd = sc.parallelize([Vectors.dense([2, 0, 0, -2]),
...                       Vectors.dense([4, 5, 0,  3]),
...                       Vectors.dense([6, 7, 0,  8])])
>>> cStats = Statistics.colStats(rdd)
>>> cStats.mean()
array([ 4.,  4.,  0.,  3.])
>>> cStats.variance()
array([  4.,  13.,   0.,  25.])
>>> cStats.count()
3
>>> cStats.numNonzeros()
array([ 3.,  2.,  0.,  3.])
>>> cStats.max()
array([ 6.,  7.,  0.,  8.])
>>> cStats.min()
array([ 2.,  0.,  0., -2.])
```

static **corr**(*x*, *y=None*, *method=None*)

Compute the correlation (matrix) for the input RDD(s) using the specified method. Methods currently supported: *pearson (default), spearman*.

If a single RDD of Vectors is passed in, a correlation matrix comparing the columns in the input RDD is returned. Use *method=* to specify the method to be used for single RDD inout. If two RDDs of floats are passed in, a single float is returned.

Parameters:
- **x** – an RDD of vector for which the correlation matrix is to be computed, or an RDD of float of the same cardinality as y when y is specified.
- **y** – an RDD of float of the same cardinality as x.
- **method** – String specifying the method to use for computing correlation. Supported: *pearson* (default), *spearman*

Returns: Correlation matrix comparing columns in x.

```
>>> x = sc.parallelize([1.0, 0.0, -2.0], 2)
>>> y = sc.parallelize([4.0, 5.0, 3.0], 2)
>>> zeros = sc.parallelize([0.0, 0.0, 0.0], 2)
>>> abs(Statistics.corr(x, y) - 0.6546537) < 1e-7
True
>>> Statistics.corr(x, y) == Statistics.corr(x, y, "pearson")
True
>>> Statistics.corr(x, y, "spearman")
```

```
0.5
>>> from math import isnan
>>> isnan(Statistics.corr(x, zeros))
True
>>> from pyspark.mllib.linalg import Vectors
>>> rdd = sc.parallelize([Vectors.dense([1, 0, 0, -2]), Vectors.dense([4, 5, 0, 3]),
...                       Vectors.dense([6, 7, 0,  8]), Vectors.dense([9, 0, 0, 1])])
>>> pearsonCorr = Statistics.corr(rdd)
>>> print(str(pearsonCorr).replace('nan', 'NaN'))
[[ 1.          0.05564149         NaN  0.40047142]
 [ 0.05564149  1.                 NaN  0.91359586]
 [        NaN         NaN  1.                 NaN]
 [ 0.40047142  0.91359586         NaN  1.        ]]
>>> spearmanCorr = Statistics.corr(rdd, method="spearman")
>>> print(str(spearmanCorr).replace('nan', 'NaN'))
[[ 1.          0.10540926         NaN  0.4       ]
 [ 0.10540926  1.                 NaN  0.9486833 ]
 [        NaN         NaN  1.                 NaN]
 [ 0.4         0.9486833          NaN  1.        ]]
>>> try:
...     Statistics.corr(rdd, "spearman")
...     print("Method name as second argument without 'method=' shouldn't be allowed.")
... except TypeError:
...     pass
```

static **kolmogorovSmirnovTest**(*data*, *distName='norm'*, **params*)

Performs the Kolmogorov-Smirnov (KS) test for data sampled from a continuous distribution. It tests the null hypothesis that the data is generated from a particular distribution.

The given data is sorted and the Empirical Cumulative Distribution Function (ECDF) is calculated which for a given point is the number of points having a CDF value lesser than it divided by the total number of points.

Since the data is sorted, this is a step function that rises by (1 / length of data) for every ordered point.

The KS statistic gives us the maximum distance between the ECDF and the CDF. Intuitively if this statistic is large, the probabilty that the null hypothesis is true becomes small. For specific details of the implementation, please have a look at the Scala documentation.

Parameters:
- **data** – RDD, samples from the data
- **distName** – string, currently only “norm” is supported. (Normal distribution) to calculate the theoretical distribution of the data.
- **params** – additional values which need to be provided for a certain distribution. If not provided, the default values are used.

Returns: KolmogorovSmirnovTestResult object containing the test statistic, degrees of freedom, p-value, the method used, and the null

hypothesis.

```
>>> kstest = Statistics.kolmogorovSmirnovTest
>>> data = sc.parallelize([-1.0, 0.0, 1.0])
>>> ksmodel = kstest(data, "norm")
>>> print(round(ksmodel.pValue, 3))
1.0
>>> print(round(ksmodel.statistic, 3))
0.175
>>> ksmodel.nullHypothesis
u'Sample follows theoretical distribution'
```

```
>>> data = sc.parallelize([2.0, 3.0, 4.0])
>>> ksmodel = kstest(data, "norm", 3.0, 1.0)
>>> print(round(ksmodel.pValue, 3))
1.0
>>> print(round(ksmodel.statistic, 3))
0.175
```

class pyspark.mllib.stat.**MultivariateStatisticalSummary**(*java_model*)

Trait for multivariate statistical summary of a data matrix.

count()

max()

mean()

min()

normL1()

normL2()

numNonzeros()

variance()

class pyspark.mllib.stat.**ChiSqTestResult**(*java_model*)

Contains test results for the chi-squared hypothesis test.

method

Name of the test method

class pyspark.mllib.stat.**MultivariateGaussian**

Represents a (mu, sigma) tuple

```
>>> m = MultivariateGaussian(Vectors.dense([11,12]),DenseMatrix(2, 2, (1.0, 3.0, 5.0, 2.0)))
>>> (m.mu, m.sigma.toArray())
(DenseVector([11.0, 12.0]), array([[ 1., 5.],[ 3., 2.]]))
>>> (m[0], m[1])
(DenseVector([11.0, 12.0]), array([[ 1., 5.],[ 3., 2.]]))
```

class pyspark.mllib.stat.**KernelDensity**

Estimate probability density at required points given a RDD of samples from the population.

```
>>> kd = KernelDensity()
>>> sample = sc.parallelize([0.0, 1.0])
>>> kd.setSample(sample)
>>> kd.estimate([0.0, 1.0])
array([ 0.12938758,  0.12938758])
```

estimate(*points*)

Estimate the probability density at points

setBandwidth(*bandwidth*)

Set bandwidth of each sample. Defaults to 1.0

setSample(*sample*)

Set sample points from the population. Should be a RDD

pyspark.mllib.tree module

class pyspark.mllib.tree.**DecisionTreeModel**(*java_model*) [source]

A decision tree model for classification or regression.

New in version 1.1.0.

call(*name*, **a*)

Call method of java_model

depth() [source]

Get depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes).

New in version 1.1.0.

classmethod **load**(*sc*, *path*)

Load a model from the given path.

New in version 1.3.0.

numNodes() [source]

Get number of nodes in tree, including leaf nodes.

New in version 1.1.0.

predict(*x*) [source]

Predict the label of one or more examples.

Note: In Python, predict cannot currently be used within an RDD
transformation or action. Call predict directly on the RDD instead.

Parameters: **x** – Data point (feature vector), or an RDD of data points (feature vectors).

New in version 1.1.0.

save(*sc*, *path*)

Save this model to the given path.

New in version 1.3.0.

toDebugString() [source]

full model.

New in version 1.2.0.

class pyspark.mllib.tree.**DecisionTree** [source]

Learning algorithm for a decision tree model for classification or regression.

New in version 1.1.0.

classmethod **trainClassifier**(*data*, *numClasses*, *categoricalFeaturesInfo*, *impurity='gini'*, *maxDepth=5*, *maxBins=32*, *minInstancesPerNode=1*, *minInfoGain=0.0*) [source]

Train a decision tree model for classification.

Parameters:
- **data** – Training data: RDD of LabeledPoint. Labels should take values {0, 1, ..., numClasses-1}.
- **numClasses** – Number of classes for classification.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **impurity** – Criterion used for information gain calculation. Supported values: “gini” or “entropy”. (default: “gini”)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 5)
- **maxBins** – Number of bins used for finding splits at each node. (default: 32)
- **minInstancesPerNode** – Minimum number of instances required at child nodes to create the parent split. (default: 1)
- **minInfoGain** – Minimum info gain required to create a split. (default: 0.0)

Returns: DecisionTreeModel.

Example usage:

```
>>> from numpy import array
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import DecisionTree
>>>
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(1.0, [1.0]),
```

```
...     LabeledPoint(1.0, [2.0]),
...     LabeledPoint(1.0, [3.0])
... ]
>>> model = DecisionTree.trainClassifier(sc.parallelize(data), 2, {})
>>> print(model)
DecisionTreeModel classifier of depth 1 with 3 nodes
```

```
>>> print(model.toDebugString())
DecisionTreeModel classifier of depth 1 with 3 nodes
  If (feature 0 <= 0.0)
   Predict: 0.0
  Else (feature 0 > 0.0)
   Predict: 1.0

>>> model.predict(array([1.0]))
1.0
>>> model.predict(array([0.0]))
0.0
>>> rdd = sc.parallelize([[1.0], [0.0]])
>>> model.predict(rdd).collect()
[1.0, 0.0]
```

New in version 1.1.0.

classmethod **trainRegressor**(*data*, *categoricalFeaturesInfo*, *impurity='variance'*, *maxDepth=5*, *maxBins=32*, *minInstancesPerNode=1*, *minInfoGain=0.0*) [source]

Train a decision tree model for regression.

Parameters:
- **data** – Training data: RDD of LabeledPoint. Labels are real numbers.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **impurity** – Criterion used for information gain calculation. The only supported value for regression is “variance”. (default: “variance”)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 5)
- **maxBins** – Number of bins used for finding splits at each node. (default: 32)
- **minInstancesPerNode** – Minimum number of instances required at child nodes to create the parent split. (default: 1)
- **minInfoGain** – Minimum info gain required to create a split. (default: 0.0)

Returns: DecisionTreeModel.

Example usage:

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import DecisionTree
>>> from pyspark.mllib.linalg import SparseVector
>>>
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
... ]
>>>
>>> model = DecisionTree.trainRegressor(sc.parallelize(sparse_data), {})
>>> model.predict(SparseVector(2, {1: 1.0}))
1.0
>>> model.predict(SparseVector(2, {1: 0.0}))
0.0
>>> rdd = sc.parallelize([[0.0, 1.0], [0.0, 0.0]])
>>> model.predict(rdd).collect()
[1.0, 0.0]
```

New in version 1.1.0.

class pyspark.mllib.tree.**RandomForestModel**(*java_model*) [source]

Represents a random forest model.

New in version 1.2.0.

call(*name*, **a*)

Call method of java_model

classmethod **load**(*sc*, *path*)

Load a model from the given path.

New in version 1.3.0.

numTrees()

Get number of trees in ensemble.

New in version 1.3.0.

predict(*x*)

Predict values for a single data point or an RDD of points using the model trained.

Note: In Python, predict cannot currently be used within an RDD
transformation or action. Call predict directly on the RDD instead.

New in version 1.3.0.

save(*sc*, *path*)

Save this model to the given path.

New in version 1.3.0.

toDebugString()

Full model

New in version 1.3.0.

totalNumNodes()

Get total number of nodes, summed over all trees in the ensemble.

New in version 1.3.0.

class pyspark.mllib.tree.**RandomForest** [source]

Learning algorithm for a random forest model for classification or regression.

New in version 1.2.0.

supportedFeatureSubsetStrategies = *('auto', 'all', 'sqrt', 'log2', 'onethird')*

classmethod **trainClassifier**(*data*, *numClasses*, *categoricalFeaturesInfo*, *numTrees*, *featureSubsetStrategy='auto'*, *impurity='gini'*, *maxDepth=4*, *maxBins=32*, *seed=None*) [source]

Train a random forest model for binary or multiclass classification.

Parameters:
- **data** – Training dataset: RDD of LabeledPoint. Labels should take values {0, 1, ..., numClasses-1}.
- **numClasses** – Number of classes for classification.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **numTrees** – Number of trees in the random forest.
- **featureSubsetStrategy** – Number of features to consider for splits at each node. Supported values: “auto”, “all”, “sqrt”, “log2”, “onethird”. If “auto” is set, this parameter is set based on numTrees: if numTrees == 1, set to “all”; if numTrees > 1 (forest) set to “sqrt”. (default: “auto”)
- **impurity** – Criterion used for information gain calculation. Supported values: “gini” or “entropy”. (default: “gini”)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 4)
- **maxBins** – Maximum number of bins used for splitting features. (default: 32)
- **seed** – Random seed for bootstrapping and choosing feature subsets. Set as None to generate seed based on system time. (default: None)

Returns: RandomForestModel that can be used for prediction.

Example usage:

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import RandomForest
>>>
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(0.0, [1.0]),
...     LabeledPoint(1.0, [2.0]),
...     LabeledPoint(1.0, [3.0])
... ]
>>> model = RandomForest.trainClassifier(sc.parallelize(data), 2, {}, 3, seed=42)
>>> model.numTrees()
3
>>> model.totalNumNodes()
7
>>> print(model)
TreeEnsembleModel classifier with 3 trees

>>> print(model.toDebugString())
TreeEnsembleModel classifier with 3 trees

  Tree 0:
    Predict: 1.0
```

```
  Tree 1:
    If (feature 0 <= 1.0)
     Predict: 0.0
    Else (feature 0 > 1.0)
     Predict: 1.0
  Tree 2:
    If (feature 0 <= 1.0)
     Predict: 0.0
    Else (feature 0 > 1.0)
     Predict: 1.0

>>> model.predict([2.0])
1.0
>>> model.predict([0.0])
0.0
>>> rdd = sc.parallelize([[3.0], [1.0]])
>>> model.predict(rdd).collect()
[1.0, 0.0]
```

New in version 1.2.0.

classmethod **trainRegressor**(*data*, *categoricalFeaturesInfo*, *numTrees*, *featureSubsetStrategy='auto'*, *impurity='variance'*, *maxDepth=4*, *maxBins=32*, *seed=None*) [source]

Train a random forest model for regression.

Parameters:

- **data** – Training dataset: RDD of LabeledPoint. Labels are real numbers.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **numTrees** – Number of trees in the random forest.
- **featureSubsetStrategy** – Number of features to consider for splits at each node. Supported values: “auto”, “all”, “sqrt”, “log2”, “onethird”. If “auto” is set, this parameter is set based on numTrees: if numTrees == 1, set to “all”; if numTrees > 1 (forest) set to “onethird” for regression. (default: “auto”)
- **impurity** – Criterion used for information gain calculation. The only supported value for regression is “variance”. (default: “variance”)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 4)
- **maxBins** – Maximum number of bins used for splitting features. (default: 32)
- **seed** – Random seed for bootstrapping and choosing feature subsets. Set as None to generate seed based on system time. (default: None)

Returns: RandomForestModel that can be used for prediction.

Example usage:

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import RandomForest
>>> from pyspark.mllib.linalg import SparseVector
>>>
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {0: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(0.0, SparseVector(2, {0: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
... ]
>>>
>>> model = RandomForest.trainRegressor(sc.parallelize(sparse_data), {}, 2, seed=42)
>>> model.numTrees()
2
>>> model.totalNumNodes()
4
>>> model.predict(SparseVector(2, {1: 1.0}))
1.0
>>> model.predict(SparseVector(2, {0: 1.0}))
0.5
>>> rdd = sc.parallelize([[0.0, 1.0], [1.0, 0.0]])
>>> model.predict(rdd).collect()
[1.0, 0.5]
```

New in version 1.2.0.

class pyspark.mllib.tree.**GradientBoostedTreesModel**(*java_model*) [source]

Represents a gradient-boosted tree model.

New in version 1.3.0.

call(*name*, **a*)

Call method of java_model

classmethod **load**(*sc*, *path*)

Load a model from the given path.

New in version 1.3.0.

numTrees()

Get number of trees in ensemble.

New in version 1.3.0.

predict(*x*)

Predict values for a single data point or an RDD of points using the model trained.

Note: In Python, predict cannot currently be used within an RDD
transformation or action. Call predict directly on the RDD instead.

New in version 1.3.0.

save(*sc*, *path*)

Save this model to the given path.

New in version 1.3.0.

toDebugString()

Full model

New in version 1.3.0.

totalNumNodes()

Get total number of nodes, summed over all trees in the ensemble.

New in version 1.3.0.

class pyspark.mllib.tree.**GradientBoostedTrees** [source]

Learning algorithm for a gradient boosted trees model for classification or regression.

New in version 1.3.0.

classmethod **trainClassifier**(*data*, *categoricalFeaturesInfo*, *loss='logLoss'*, *numIterations=100*, *learningRate=0.1*, *maxDepth=3*, *maxBins=32*)
[source]

Train a gradient-boosted trees model for classification.

Parameters:
- **data** – Training dataset: RDD of LabeledPoint. Labels should take values {0, 1}.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **loss** – Loss function used for minimization during gradient boosting. Supported values: “logLoss”, “leastSquaresError”, “leastAbsoluteError”. (default: “logLoss”)
- **numIterations** – Number of iterations of boosting. (default: 100)
- **learningRate** – Learning rate for shrinking the contribution of each estimator. The learning rate should be between in the interval (0, 1]. (default: 0.1)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 3)
- **maxBins** – Maximum number of bins used for splitting features. DecisionTree requires maxBins >= max categories. (default: 32)

Returns: GradientBoostedTreesModel that can be used for prediction.

Example usage:

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import GradientBoostedTrees
>>>
>>> data = [
...     LabeledPoint(0.0, [0.0]),
...     LabeledPoint(0.0, [1.0]),
...     LabeledPoint(1.0, [2.0]),
...     LabeledPoint(1.0, [3.0])
... ]
>>>
>>> model = GradientBoostedTrees.trainClassifier(sc.parallelize(data), {}, numIterations=10)
>>> model.numTrees()
10
>>> model.totalNumNodes()
30
>>> print(model)  # it already has newline
TreeEnsembleModel classifier with 10 trees

>>> model.predict([2.0])
1.0
>>> model.predict([0.0])
0.0
```

```
>>> rdd = sc.parallelize([[2.0], [0.0]])
>>> model.predict(rdd).collect()
[1.0, 0.0]
```

New in version 1.3.0.

classmethod **trainRegressor**(*data*, *categoricalFeaturesInfo*, *loss='leastSquaresError'*, *numIterations=100*, *learningRate=0.1*, *maxDepth=3*, *maxBins=32*) [source]

Train a gradient-boosted trees model for regression.

Parameters:
- **data** – Training dataset: RDD of LabeledPoint. Labels are real numbers.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. An entry (n -> k) indicates that feature n is categorical with k categories indexed from 0: {0, 1, ..., k-1}.
- **loss** – Loss function used for minimization during gradient boosting. Supported values: “logLoss”, “leastSquaresError”, “leastAbsoluteError”. (default: “leastSquaresError”)
- **numIterations** – Number of iterations of boosting. (default: 100)
- **learningRate** – Learning rate for shrinking the contribution of each estimator. The learning rate should be between in the interval (0, 1]. (default: 0.1)
- **maxDepth** – Maximum depth of tree (e.g. depth 0 means 1 leaf node, depth 1 means 1 internal node + 2 leaf nodes). (default: 3)
- **maxBins** – Maximum number of bins used for splitting features. DecisionTree requires maxBins >= max categories. (default: 32)

Returns: GradientBoostedTreesModel that can be used for prediction.

Example usage:

```
>>> from pyspark.mllib.regression import LabeledPoint
>>> from pyspark.mllib.tree import GradientBoostedTrees
>>> from pyspark.mllib.linalg import SparseVector
>>>
>>> sparse_data = [
...     LabeledPoint(0.0, SparseVector(2, {0: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
...     LabeledPoint(0.0, SparseVector(2, {0: 1.0})),
...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
... ]
>>>
>>> data = sc.parallelize(sparse_data)
```

```
>>> model = GradientBoostedTrees.trainRegressor(data, {}, numIterations=10)
>>> model.numTrees()
10
>>> model.totalNumNodes()
12
>>> model.predict(SparseVector(2, {1: 1.0}))
1.0
>>> model.predict(SparseVector(2, {0: 1.0}))
0.0
>>> rdd = sc.parallelize([[0.0, 1.0], [1.0, 0.0]])
>>> model.predict(rdd).collect()
[1.0, 0.0]
```

New in version 1.3.0.

pyspark.mllib.util module

class pyspark.mllib.util.**JavaLoader** [source]

Mixin for classes which can load saved models using its Scala implementation.

New in version 1.3.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path.

New in version 1.3.0.

class pyspark.mllib.util.**JavaSaveable** [source]

Mixin for models that provide save() through their Scala implementation.

New in version 1.3.0.

save(*sc*, *path*) [source]

Save this model to the given path.

New in version 1.3.0.

class pyspark.mllib.util.**LinearDataGenerator** [source]

Utils for generating linear data.

New in version 1.5.0.

static **generateLinearInput**(*intercept*, *weights*, *xMean*, *xVariance*, *nPoints*, *seed*, *eps*) [source]

Param: intercept bias factor, the term c in X'w + c
Param: weights feature vector, the term w in X'w + c
Param: xMean Point around which the data X is centered.
Param: xVariance Variance of the given data
Param: nPoints Number of points to be generated
Param: seed Random Seed
Param: eps Used to scale the noise. If eps is set high, the amount of gaussian noise added is more.

Returns a list of LabeledPoints of length nPoints

New in version 1.5.0.

static **generateLinearRDD**(*sc*, *nexamples*, *nfeatures*, *eps*, *nParts=2*, *intercept=0.0*) [source]

Generate a RDD of LabeledPoints.

New in version 1.5.0.

class pyspark.mllib.util.**Loader** [source]

Mixin for classes which can load saved models from files.

New in version 1.3.0.

classmethod **load**(*sc*, *path*) [source]

Load a model from the given path. The model should have been saved using py:meth:*Saveable.save*.

Parameters:
- **sc** – Spark context used for loading model files.
- **path** – Path specifying the directory to which the model was saved.

Returns: model instance

class pyspark.mllib.util.**MLUtils** [source]

Helper methods to load, save and pre-process data used in MLlib.

New in version 1.0.0.

static **appendBias**(*data*) [source]

Returns a new vector with *1.0* (bias) appended to the end of the input vector.

New in version 1.5.0.

static **convertMatrixColumnsFromML**(*dataset*, **cols*) [source]

Converts matrix columns in an input DataFrame to the **pyspark.mllib.linalg.Matrix** type from the new **pyspark.ml.linalg.Matrix** type under the *spark.ml* package.

Parameters:
- **dataset** – input dataset
- **cols** – a list of matrix columns to be converted. Old matrix columns will be ignored. If unspecified, all new matrix columns will be converted except nested ones.

Returns: the input dataset with new matrix columns converted to the old matrix type

```
>>> import pyspark
>>> from pyspark.ml.linalg import Matrices
>>> from pyspark.mllib.util import MLUtils
>>> df = spark.createDataFrame(
...     [(0, Matrices.sparse(2, 2, [0, 2, 3], [0, 1, 1], [2, 3, 4]),
...     Matrices.dense(2, 2, range(4)))], ["id", "x", "y"])
>>> r1 = MLUtils.convertMatrixColumnsFromML(df).first()
>>> isinstance(r1.x, pyspark.mllib.linalg.SparseMatrix)
True
>>> isinstance(r1.y, pyspark.mllib.linalg.DenseMatrix)
True
>>> r2 = MLUtils.convertMatrixColumnsFromML(df, "x").first()
>>> isinstance(r2.x, pyspark.mllib.linalg.SparseMatrix)
True
>>> isinstance(r2.y, pyspark.ml.linalg.DenseMatrix)
True
```

New in version 2.0.0.

static **convertMatrixColumnsToML**(*dataset*, **cols*) [source]

Converts matrix columns in an input DataFrame from the **pyspark.mllib.linalg.Matrix** type to the new **pyspark.ml.linalg.Matrix** type under the *spark.ml* package.

Parameters:
- **dataset** – input dataset
- **cols** – a list of matrix columns to be converted. New matrix columns will be ignored. If unspecified, all old matrix columns will be converted excepted nested ones.

Returns: the input dataset with old matrix columns converted to the new matrix type

```
>>> import pyspark
>>> from pyspark.mllib.linalg import Matrices
>>> from pyspark.mllib.util import MLUtils
>>> df = spark.createDataFrame(
...     [(0, Matrices.sparse(2, 2, [0, 2, 3], [0, 1, 1], [2, 3, 4]),
...     Matrices.dense(2, 2, range(4)))], ["id", "x", "y"])
>>> r1 = MLUtils.convertMatrixColumnsToML(df).first()
>>> isinstance(r1.x, pyspark.ml.linalg.SparseMatrix)
True
>>> isinstance(r1.y, pyspark.ml.linalg.DenseMatrix)
True
>>> r2 = MLUtils.convertMatrixColumnsToML(df, "x").first()
>>> isinstance(r2.x, pyspark.ml.linalg.SparseMatrix)
True
>>> isinstance(r2.y, pyspark.mllib.linalg.DenseMatrix)
True
```

New in version 2.0.0.

static **convertVectorColumnsFromML**(*dataset*, **cols*) [source]

Converts vector columns in an input DataFrame to the **pyspark.mllib.linalg.Vector** type from the new **pyspark.ml.linalg.Vector** type under the *spark.ml* package.

Parameters:
- **dataset** – input dataset
- **cols** – a list of vector columns to be converted. Old vector columns will be ignored. If unspecified, all new vector columns will be converted except nested ones.

Returns: the input dataset with new vector columns converted to the old vector type

```
>>> import pyspark
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.mllib.util import MLUtils
>>> df = spark.createDataFrame(
...     [(0, Vectors.sparse(2, [1], [1.0]), Vectors.dense(2.0, 3.0))],
...     ["id", "x", "y"])
```

```
>>> r1 = MLUtils.convertVectorColumnsFromML(df).first()
>>> isinstance(r1.x, pyspark.mllib.linalg.SparseVector)
True
>>> isinstance(r1.y, pyspark.mllib.linalg.DenseVector)
True
>>> r2 = MLUtils.convertVectorColumnsFromML(df, "x").first()
>>> isinstance(r2.x, pyspark.mllib.linalg.SparseVector)
True
>>> isinstance(r2.y, pyspark.ml.linalg.DenseVector)
True
```

New in version 2.0.0.

static **convertVectorColumnsToML**(*dataset*, **cols*) [source]

Converts vector columns in an input DataFrame from the **pyspark.mllib.linalg.Vector** type to the new **pyspark.ml.linalg.Vector** type under the *spark.ml* package.

Parameters:
- **dataset** – input dataset
- **cols** – a list of vector columns to be converted. New vector columns will be ignored. If unspecified, all old vector columns will be converted excepted nested ones.

Returns: the input dataset with old vector columns converted to the new vector type

```
>>> import pyspark
>>> from pyspark.mllib.linalg import Vectors
>>> from pyspark.mllib.util import MLUtils
>>> df = spark.createDataFrame(
...     [(0, Vectors.sparse(2, [1], [1.0]), Vectors.dense(2.0, 3.0))],
...     ["id", "x", "y"])
>>> r1 = MLUtils.convertVectorColumnsToML(df).first()
>>> isinstance(r1.x, pyspark.ml.linalg.SparseVector)
True
>>> isinstance(r1.y, pyspark.ml.linalg.DenseVector)
True
>>> r2 = MLUtils.convertVectorColumnsToML(df, "x").first()
>>> isinstance(r2.x, pyspark.ml.linalg.SparseVector)
True
>>> isinstance(r2.y, pyspark.mllib.linalg.DenseVector)
True
```

New in version 2.0.0.

static **loadLabeledPoints**(*sc*, *path*, *minPartitions=None*) [source]

Load labeled points saved using RDD.saveAsTextFile.

Parameters:
- **sc** – Spark context
- **path** – file or directory path in any Hadoop-supported file system URI
- **minPartitions** – min number of partitions

Returns: labeled data stored as an RDD of LabeledPoint

```
>>> from tempfile import NamedTemporaryFile
>>> from pyspark.mllib.util import MLUtils
>>> from pyspark.mllib.regression import LabeledPoint
>>> examples = [LabeledPoint(1.1, Vectors.sparse(3, [(0, -1.23), (2, 4.56e-7)])),                    LabeledPoint(0.0, Vecto
>>> tempFile = NamedTemporaryFile(delete=True)
>>> tempFile.close()
>>> sc.parallelize(examples, 1).saveAsTextFile(tempFile.name)
>>> MLUtils.loadLabeledPoints(sc, tempFile.name).collect()
[LabeledPoint(1.1, (3,[0,2],[-1.23,4.56e-07])), LabeledPoint(0.0, [1.01,2.02,3.03])]
```

New in version 1.1.0.

static **loadLibSVMFile**(*sc*, *path*, *numFeatures=-1*, *minPartitions=None*, *multiclass=None*) [source]

Loads labeled data in the LIBSVM format into an RDD of LabeledPoint. The LIBSVM format is a text-based format used by LIBSVM and LIBLINEAR. Each line represents a labeled sparse feature vector using the following format:

label index1:value1 index2:value2 ...

where the indices are one-based and in ascending order. This method parses each line into a LabeledPoint, where the feature indices are converted to zero-based.

Parameters:
- **sc** – Spark context
- **path** – file or directory path in any Hadoop-supported file system URI
- **numFeatures** – number of features, which will be determined from the input data if a nonpositive value is given. This is useful when the dataset is already split into multiple files and you want to load them separately, because some features may not present in certain files, which leads to inconsistent feature dimensions.
- **minPartitions** – min number of partitions

Returns: labeled data stored as an RDD of LabeledPoint

```
>>> from tempfile import NamedTemporaryFile
>>> from pyspark.mllib.util import MLUtils
>>> from pyspark.mllib.regression import LabeledPoint
>>> tempFile = NamedTemporaryFile(delete=True)
>>> _ = tempFile.write(b"+1 1:1.0 3:2.0 5:3.0\n-1\n-1 2:4.0 4:5.0 6:6.0")
>>> tempFile.flush()
>>> examples = MLUtils.loadLibSVMFile(sc, tempFile.name).collect()
>>> tempFile.close()
>>> examples[0]
LabeledPoint(1.0, (6,[0,2,4],[1.0,2.0,3.0]))
>>> examples[1]
LabeledPoint(-1.0, (6,[],[]))
>>> examples[2]
LabeledPoint(-1.0, (6,[1,3,5],[4.0,5.0,6.0]))
```

New in version 1.0.0.

static **loadVectors**(*sc*, *path*) [source]

Loads vectors saved using *RDD[Vector].saveAsTextFile* with the default number of partitions.

New in version 1.5.0.

static **saveAsLibSVMFile**(*data*, *dir*) [source]

Save labeled data in LIBSVM format.

Parameters:
- **data** – an RDD of LabeledPoint to be saved
- **dir** – directory to save the data

```
>>> from tempfile import NamedTemporaryFile
>>> from fileinput import input
>>> from pyspark.mllib.regression import LabeledPoint
>>> from glob import glob
>>> from pyspark.mllib.util import MLUtils
>>> examples = [LabeledPoint(1.1, Vectors.sparse(3, [(0, 1.23), (2, 4.56)])),                    LabeledPoint(0.0, Vectors.d
>>> tempFile = NamedTemporaryFile(delete=True)
>>> tempFile.close()
>>> MLUtils.saveAsLibSVMFile(sc.parallelize(examples), tempFile.name)
>>> ''.join(sorted(input(glob(tempFile.name + "/part-0000*"))))
```

```
'0.0 1:1.01 2:2.02 3:3.03\n1.1 1:1.23 3:4.56\n'
```

New in version 1.0.0.

class pyspark.mllib.util.**Saveable** [source]

Mixin for models and transformers which may be saved as files.

New in version 1.3.0.

save(*sc*, *path*) [source]

Save this model to the given path.

This saves:

- human-readable (JSON) model metadata to path/metadata/
- Parquet formatted data to path/data/

The model may be loaded using py:meth:*Loader.load*.

Parameters:
- **sc** – Spark context used to save model data.
- **path** – Path specifying the directory in which to save this model. If the directory already exists, this method throws an exception.