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 be a 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
>>> 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
```

clearThreshold()

> **Note:** Experimental

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

intercept

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

numClasses [source]

numFeatures [source]

predict(*x*) [source]

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

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

setThreshold(*value*)

> **Note:** Experimental

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 an positive, and negative otherwise. It is used for binary classification only.

threshold

> **Note:** Experimental

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

weights

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

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *miniBatchFraction=1.0*, *initialWeights=None*, *regParam=0.01*, *regType='l2'*, *intercept=False*, *validateData=True*) [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.
  Allowed values:
  - "l1" for using L1 regularization
  - "l2" for using L2 regularization
  - None for no regularization

  (default: "l2")
- **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)

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

classmethod **train**(*data*, *iterations=100*, *initialWeights=None*, *regParam=0.01*, *regType='l2'*, *intercept=False*, *corrections=10*, *tolerance=0.0001*, *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: 0.01).
- **regType** –
  The type of regularizer used for training our model.
  Allowed values:
  - "l1" for using L1 regularization
  - "l2" for using L2 regularization
  - None for no regularization

  (default: "l2")

- **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 (default: 10).
- **tolerance** – The convergence tolerance of iterations for L-BFGS (default: 1e-4).
- **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=
>>> lrm.predict([1.0, 0.0])
1
>>> lrm.predict([0.0, 1.0])
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
```

clearThreshold()

> **Note:** Experimental

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

intercept

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

predict(*x*) [source]

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

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

setThreshold(*value*)

> **Note:** Experimental

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 an positive, and negative otherwise. It is used for binary classification only.

threshold

> **Note:** Experimental

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

weights

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

classmethod **train**(*data*, *iterations=100*, *step=1.0*, *regParam=0.01*, *miniBatchFraction=1.0*, *initialWeights=None*, *regType='l2'*, *intercept=False*, *validateData=True*) [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
  - None for no regularization

  (default: "l2")
- **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)

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,
True
>>> from shutil import rmtree
>>> try:
...     rmtree(path)
... except OSError:
...     pass
```

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

predict(*x*) [source]

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

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.

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

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).

pyspark.mllib.clustering module

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, runs=30, initializationMode="r
...                       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-me
...                                       seed=50, initializationSteps=5, epsilon=1
>>> 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
```

clusterCenters [source]

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

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.

k [source]

Total number of clusters.

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

predict(*x*) [source]

Find the cluster to which x belongs in this model.

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.

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

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

Train a k-means clustering model.

class pyspark.mllib.clustering.**GaussianMixtureModel**(*weights*, *gaussians*) [source]

A clustering model derived from the Gaussian Mixture Model method.

```
>>> from pyspark.mllib.linalg import Vectors, DenseMatrix
>>> 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,
>>> 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]
True
>>> labels[4]==labels[5]
True
>>> 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]==labels[2]
True
>>> labels[3]==labels[4]
True
>>> clusterdata_3 = sc.parallelize(data.reshape(15, 1))
>>> im = GaussianMixtureModel([0.5, 0.5],
...      [MultivariateGaussian(Vectors.dense([-1.0]), DenseMatrix(1, 1, [1.0])),
...      MultivariateGaussian(Vectors.dense([1.0]), DenseMatrix(1, 1, [1.0]))])
>>> model = GaussianMixture.train(clusterdata_3, 2, initialModel=im)
```

gaussians [source]

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

k [source]

Number of gaussians in mixture.

predict(*x*) [source]

Find the cluster to which the points in 'x' has maximum membership in this model.

Parameters: **x** – RDD of data points.
Returns: cluster_labels. RDD of cluster labels.

predictSoft(*x*) [source]

Find the membership of each point in 'x' to all mixture components.

Parameters: **x** – RDD of data points.
Returns: membership_matrix. RDD of array of double values.

weights [source]

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

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

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

Parameters:
- **data** – RDD of data points
- **k** – Number of components
- **convergenceTol** – Threshold value to check the convergence criteria. Defaults to 1e-3
- **maxIterations** – Number of iterations. Default to 100
- **seed** – Random Seed
- **initialModel** – GaussianMixtureModel for initializing learning

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

Train a Gaussian Mixture clustering model.

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), (
>>> metrics = BinaryClassificationMetrics(scoreAndLabels)
>>> metrics.areaUnderROC
0.70...
>>> metrics.areaUnderPR
0.83...
>>> metrics.unpersist()
```

areaUnderPR [source]

Computes the area under the precision-recall curve.

areaUnderROC [source]

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

unpersist() [source]

Unpersists intermediate RDDs used in the computation.

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
0.95...
>>> metrics.meanAbsoluteError
0.5...
>>> metrics.meanSquaredError
0.37...
>>> metrics.rootMeanSquaredError
0.61...
>>> metrics.r2
0.94...
```

explainedVariance [source]

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

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.

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.

r2 [source]

Returns R^2^, the coefficient of determination.

rootMeanSquaredError [source]

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

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

Evaluator for multiclass classification.

:param 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.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.precision()
0.66...
>>> metrics.recall()
0.66...
>>> metrics.weightedFalsePositiveRate
0.19...
>>> metrics.weightedPrecision
0.68...
>>> metrics.weightedRecall
0.66...
>>> metrics.weightedFMeasure()
0.66...
>>> metrics.weightedFMeasure(2.0)
0.65...
```

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

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

falsePositiveRate(*label*) [source]

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

precision(*label=None*) [source]

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

recall(*label=None*) [source]

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

truePositiveRate(*label*) [source]

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

weightedFMeasure(*beta=None*) [source]

Returns weighted averaged f-measure.

weightedFalsePositiveRate [source]

Returns weighted false positive rate.

weightedPrecision [source]

Returns weighted averaged precision.

weightedRecall [source]

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

weightedTruePositiveRate [source]

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

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...
```

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.

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.

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.

pyspark.mllib.feature module

Python package for feature in MLlib.

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

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

Note: Experimental

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])
```

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.

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

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

> **Note:** Experimental

Represents a StandardScaler model that can transform vectors.

setWithMean(*withMean*) [source]

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

setWithStd(*withStd*) [source]

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

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.

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

Bases: **object**

> **Note:** Experimental

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])
```

fit(*dataset*) [source]

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

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

Returns: a StandardScalarModel

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

Bases: **object**

> **Note:** Experimental

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, {...})
```

indexOf(*term*) [source]

Returns the index of the input term.

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.

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

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

Represents an IDF model that can transform term frequency vectors.

idf() [source]

Returns the current IDF vector.

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

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

Bases: **object**

> **Note:** Experimental

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})
```

fit(*dataset*) [source]

Computes the inverse document frequency.

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

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']
```

fit(*data*) [source]

Computes the vector representation of each word in vocabulary.

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

setLearningRate(*learningRate*) [source]

Sets initial learning rate (default: 0.025).

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).

setNumIterations(*numIterations*) [source]

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

setNumPartitions(*numPartitions*) [source]

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

setSeed(*seed*) [source]

Sets random seed.

setVectorSize(*vectorSize*) [source]

Sets vector size (default: 100).

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

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

class for Word2Vec model

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

getVectors() [source]

Returns a map of words to their vector representations.

transform(*word*) [source]

Transforms a word to its vector representation

Note: local use only

Parameters: **word** – a word

Returns: vector representation of word(s)

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

Bases: **object**

Note: Experimental

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])
```

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.

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

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

Note: Experimental

Represents a Chi Squared selector model.

transform(*vector*) [source]

Applies transformation on a vector.

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

pyspark.mllib.fpm module

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

> **Note:** Experimental

A Parallel FP-growth algorithm to mine frequent itemsets.

class **FreqItemset** [source]

Represents an (items, freq) tuple.

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 (default: same as input data).

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

> **Note:** Experimental

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), ...
```

freqItemsets() [source]

Returns the frequent itemsets of this model.

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**

toArray() [source]

Convert the vector into an numpy.ndarray :return: 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])
```

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]

Calculte the norm of a DenseVector.

```
>>> a = DenseVector([0, -1, 2, -3])
>>> a.norm(2)
3.7...
>>> a.norm(1)
6.0
```

numNonzeros() [source]

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]

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.

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, 4], [1.0, 2.0])
>>> 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
```

norm(*p*) [source]

Calculte the norm of a SparseVector.

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

numNonzeros() [source]

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

Parse string representation back into the DenseVector.

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

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, 4], [1.0, 2.0])
>>> a.squared_distance(b)
30.0
>>> b.squared_distance(a)
30.0
>>> 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.

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. Always returns a NumPy array.

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

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 tupes, 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**

Represents a local matrix.

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.

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.

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

Create a SparseMatrix

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.

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
```

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
```

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
```

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, see
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - expMean) < 0.1
True
>>> abs(mat.std() - expStd) < 0.1
True
```

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
```

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).collec
>>> mat = np.matrix(m)
>>> mat.shape
(100, 100)
>>> abs(mat.mean() - expMean) < 0.1
True
>>> abs(mat.std() - expStd) < 0.1
True
```

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
```

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
```

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
```

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
```

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
```

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
```

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
True
```

```
>>> model.productFeatures().collect()
[(1, array('d', [...])), (2, array('d', [...]))]
```

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

```
>>> model = ALS.train(ratings, 1, nonnegative=True, seed=10)
>>> model.predict(2, 2)
3.8...
```

```
>>> df = sqlContext.createDataFrame([Rating(1, 1, 1.0), Rating(1, 2, 2.0), Rating
>>> model = ALS.train(df, 1, nonnegative=True, seed=10)
>>> model.predict(2, 2)
3.8...
```

```
>>> 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
```

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

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

Predicts rating for the given user and product.

predictAll(*user_product*) [source]

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

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.

rank [source]

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.

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.

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.

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

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

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

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)
```

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.

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.

intercept [source]

weights [source]

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
>>> 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
...    miniBatchFraction=1.0, initialWeights=array([1.0]), regParam=0.1, regType=
...    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
```

intercept

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

predict(*x*)

Predict the value of the dependent variable given a vector x containing values for the independent variables.

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

weights

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

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

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

f(weights) = 1/n ||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.
  Allowed values:
  - “l1” for using L1 regularization (lasso),
  - “l2” for using L2 regularization (ridge),
  - None for no regularization

(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)

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
>>> 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=
...     regParam=0.01, miniBatchFraction=1.0, initialWeights=array([1.0]), interc
...     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
```

intercept

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

predict(*x*)

Predict the value of the dependent variable given a vector x containing values for the independent variables.

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

weights

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

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

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=
```

```
>>> 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
>>> 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]), interc
...     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
```

intercept

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

predict(*x*)

Predict the value of the dependent variable given a vector x containing values for the independent variables.

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

weights

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

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

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), (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
```

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

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.

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

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

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

Train a isotonic regression model on the given data.

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

pyspark.mllib.stat module

Python package for statistical functions in MLlib.

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

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

Note: Experimental

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,
>>> 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,
...                       Vectors.dense([6, 7, 0,  8]), Vectors.dense([9, 0,
>>> 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
... except TypeError:
...     pass
```

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*)

> **Note:** Experimental

Object containing the test results for the chi-squared hypothesis test.

degreesOfFreedom

Returns the degree(s) of freedom of the hypothesis test. Return type should be Number(e.g. Int, Double) or tuples of Numbers.

method

Name of the test method

nullHypothesis

Null hypothesis of the test.

pValue

The probability of obtaining a test statistic result at least as extreme as the one that was actually observed, assuming that the null hypothesis is true.

statistic

Test statistic.

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

Represents a (mu, sigma) tuple

```
>>> m = MultivariateGaussian(Vectors.dense([11,12]),DenseMatrix(2, 2, (1.0, 3.0,
>>> (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.]]))
```

pyspark.mllib.tree module

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

> **Note:** Experimental

A decision tree model for classification or regression.

call(*name*, **a*)

Call method of java_model

depth() [source]

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

numNodes() [source]

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).

save(*sc*, *path*)

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.

toDebugString() [source]

full model.

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

> **Note:** Experimental

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

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

Train a DecisionTreeModel for classification.

Parameters:
- **data** – Training data: RDD of LabeledPoint. Labels are integers {0,1,...,numClasses}.
- **numClasses** – Number of classes for classification.
- **categoricalFeaturesInfo** – Map from categorical feature index to number of categories. Any feature not in this map is treated as continuous.
- **impurity** – Supported values: “entropy” or “gini”
- **maxDepth** – Max depth of tree. E.g., depth 0 means 1 leaf node. Depth 1 means 1 internal node + 2 leaf nodes.
- **maxBins** – Number of bins used for finding splits at each node.
- **minInstancesPerNode** – Min number of instances required at child nodes to create the parent split
- **minInfoGain** – Min info gain required to create a split

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]
```

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

Train a DecisionTreeModel for regression.

Parameters:
- **data** – Training data: RDD of LabeledPoint. Labels are real numbers.
- **categoricalFeaturesInfo** – Map from categorical feature index to number of categories. Any feature not in this map is treated as continuous.
- **impurity** – Supported values: “variance”
- **maxDepth** – Max depth of tree. E.g., depth 0 means 1 leaf node. Depth 1 means 1 internal node + 2 leaf nodes.
- **maxBins** – Number of bins used for finding splits at each node.
- **minInstancesPerNode** – Min number of instances required at child nodes to create the parent split
- **minInfoGain** – Min info gain required to create a split

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]
```

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

> **Note:** Experimental

Represents a random forest model.

call(*name*, **a*)

Call method of java_model

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

numTrees()

Get number of trees in ensemble.

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.

save(*sc*, *path*)

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.

toDebugString()

Full model

totalNumNodes()

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

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

> **Note:** Experimental

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

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

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

Method to train a decision tree model for binary or multiclass classification. [source]

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. E.g., 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: “auto” (default), “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”.
- **impurity** – Criterion used for information gain calculation. Supported values: “gini” (recommended) or “entropy”.
- **maxDepth** – Maximum depth of the 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.

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
>>> 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]
```

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

Method to train a decision tree model for regression.

Parameters:
- **data** – Training dataset: RDD of LabeledPoint. Labels are real numbers.
- **categoricalFeaturesInfo** – Map storing arity of categorical features. E.g., 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: “auto” (default), “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.
- **impurity** – Criterion used for information gain calculation. Supported values: “variance”.
- **maxDepth** – Maximum depth of the 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.

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, s
>>> 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]
```

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

> **Note:** Experimental

Represents a gradient-boosted tree model.

call(*name*, **a*)

Call method of java_model

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

numTrees()

Get number of trees in ensemble.

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.

save(*sc*, *path*)

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.

toDebugString()

Full model

totalNumNodes()

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

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

> **Note:** Experimental

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

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

Method to 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. E.g., 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: {“logLoss” (default), “leastSquaresError”, “leastAbsoluteError”}.
- **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 the 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 (default: 32) DecisionTree requires maxBins >= max categories

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), {}, nu
>>> 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]
```

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

Method to 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. E.g., 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: {“logLoss” (default), “leastSquaresError”, “leastAbsoluteError”}.
- **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)
- **maxBins** – maximum number of bins used for splitting features (default: 32) DecisionTree requires maxBins >= max categories
- **maxDepth** – Maximum depth of the tree. E.g., depth 0 means 1 leaf node; depth 1 means 1 internal node + 2 leaf nodes. (default: 3)

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]
```

pyspark.mllib.util module

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

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

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

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

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

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.

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

Mixin for classes which can load saved models from files.

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.

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)
>>> 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,
```

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

Loads labeled data in the LIBSVM format into an RDD of LabeledPoint. The [source]
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]))
```

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)])),
>>> 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'
```

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

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

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.