Evaluation Metrics - RDD-based API

`spark.mllib` comes with a number of machine learning algorithms that can be used to learn from and make predictions on data. When these algorithms are applied to build machine learning models, there is a need to evaluate the performance of the model on some criteria, which depends on the application and its requirements. `spark.mllib` also provides a suite of metrics for the purpose of evaluating the performance of machine learning models.

Specific machine learning algorithms fall under broader types of machine learning applications like classification, regression, clustering, etc. Each of these types have well established metrics for performance evaluation and those metrics that are currently available in `spark.mllib` are detailed in this section.

Classification model evaluation

While there are many different types of classification algorithms, the evaluation of classification models all share similar principles. In a supervised classification problem, there exists a true output and a model-generated predicted output for each data point. For this reason, the results for each data point can be assigned to one of four categories:

- True Positive (TP) - label is positive and prediction is also positive
- True Negative (TN) - label is negative and prediction is also negative
- False Positive (FP) - label is negative but prediction is positive
- False Negative (FN) - label is positive but prediction is negative

These four numbers are the building blocks for most classifier evaluation metrics. A fundamental point when considering classifier evaluation is that pure accuracy (i.e. was the prediction correct or incorrect) is not generally a good metric. The reason for this is because a dataset may be highly unbalanced. For example, if a model is designed to predict fraud from a dataset where 95% of the data points are *not fraud* and 5% of the data points are *fraud*, then a naive classifier that predicts *not fraud*, regardless of input, will be 95% accurate. For this reason, metrics like precision and recall are typically used because they take into account the *type* of error. In most applications there is some desired balance between precision and recall, which can be captured by combining the two into a single metric, called the F-measure.

Binary classification

Binary classifiers are used to separate the elements of a given dataset into one of two possible groups (e.g. fraud or not fraud) and is a special case of multiclass classification. Most binary classification metrics can be generalized to multiclass classification metrics.

Threshold tuning

It is import to understand that many classification models actually output a “score” (often times a probability) for each class, where a higher score indicates higher likelihood. In the binary case, the model may output a probability for each class: $P(Y = 1|X)$ and $P(Y = 0|X)$. Instead of simply taking the higher probability, there may be some cases where the model might need to be tuned so that it only predicts a class when the probability is very high (e.g. only block a credit card transaction if the model predicts fraud with >90% probability). Therefore, there is a prediction *threshold* which determines what the predicted class will be based on the probabilities that the model outputs.

Tuning the prediction threshold will change the precision and recall of the model and is an important part of model optimization. In order to visualize how precision, recall, and other metrics change as a function of the threshold it is common practice to plot competing metrics against one another, parameterized by threshold. A P-R curve plots (precision, recall) points for different threshold values, while a receiver operating characteristic, or ROC, curve plots (recall, false positive rate) points.

Available metrics

Metric	Definition
Precision (Positive Predictive Value)	$PPV = \frac{TP}{TP+FP}$
Recall (True Positive Rate)	$TPR = \frac{TP}{P} = \frac{TP}{TP+FN}$
F-measure	$F(\beta) = \left(1+\beta^2\right) \cdot \left(\frac{PPV \cdot TPR}{\beta^2 \cdot PPV + TPR}\right)$
Receiver Operating Characteristic (ROC)	$FPR(T) = \int_T^{\infty} P_0(T)\,dT$ $TPR(T) = \int_T^{\infty} P_1(T)\,dT$
Area Under ROC Curve	$AUROC = \int_0^1 \frac{TP}{P} d\left(\frac{FP}{N}\right)$
Area Under Precision-Recall Curve	$AUPRC = \int_0^1 \frac{TP}{TP+FP} d\left(\frac{TP}{P}\right)$

Examples

Scala | Java | Python

The following code snippets illustrate how to load a sample dataset, train a binary classification algorithm on the data, and evaluate the performance of the algorithm by several binary evaluation metrics.
Refer to the `BinaryClassificationMetrics` Python docs and `LogisticRegressionWithLBFGS` Python docs for more details on the API.

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.util import MLUtils

# Several of the methods available in scala are currently missing from pyspark
# Load training data in LIBSVM format
data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_binary_classification_data.txt")

# Split data into training (60%) and test (40%)
training, test = data.randomSplit([0.6, 0.4], seed=11)
```

```
training.cache()

# Run training algorithm to build the model
model = LogisticRegressionWithLBFGS.train(training)

# Compute raw scores on the test set
predictionAndLabels = test.map(lambda lp: (float(model.predict(lp.features)), lp.label))

# Instantiate metrics object
metrics = BinaryClassificationMetrics(predictionAndLabels)

# Area under precision-recall curve
print("Area under PR = %s" % metrics.areaUnderPR)

# Area under ROC curve
print("Area under ROC = %s" % metrics.areaUnderROC)
```

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

Multiclass classification

A multiclass classification describes a classification problem where there are $M > 2$ possible labels for each data point (the case where $M = 2$ is the binary classification problem). For example, classifying handwriting samples to the digits 0 to 9, having 10 possible classes.

For multiclass metrics, the notion of positives and negatives is slightly different. Predictions and labels can still be positive or negative, but they must be considered under the context of a particular class. Each label and prediction take on the value of one of the multiple classes and so they are said to be positive for their particular class and negative for all other classes. So, a true positive occurs whenever the prediction and the label match, while a true negative occurs when neither the prediction nor the label take on the value of a given class. By this convention, there can be multiple true negatives for a given data sample. The extension of false negatives and false positives from the former definitions of positive and negative labels is straightforward.

Label based metrics

Opposed to binary classification where there are only two possible labels, multiclass classification problems have many possible labels and so the concept of label-based metrics is introduced. Accuracy measures precision across all labels - the number of times any class was predicted correctly (true positives) normalized by the number of data points. Precision by label considers only one class, and measures the number of time a specific label was predicted correctly normalized by the number of times that label appears in the output.

Available metrics

Define the class, or label, set as

$$L = \{\ell_0, \ell_1, \ldots, \ell_{M-1}\}$$

The true output vector $\mathbf{y}$ consists of N elements

$$\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{N-1} \in L$$

A multiclass prediction algorithm generates a prediction vector $\hat{\mathbf{y}}$ of N elements

$$\hat{\mathbf{y}}_0, \hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_{N-1} \in L$$

For this section, a modified delta function $\hat{\delta}(x)$ will prove useful

$$\hat{\delta}(x) = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Metric	Definition
Confusion Matrix	$C_{ij} = \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_i) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_j)$ $\begin{pmatrix} \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_1) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_1) & \dots & \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_1) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_N) \\ \vdots & \ddots & \vdots \\ \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_N) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_1) & \dots & \sum_{k=0}^{N-1} \hat{\delta}(\mathbf{y}_k - \ell_N) \cdot \hat{\delta}(\hat{\mathbf{y}}_k - \ell_N) \end{pmatrix}$
Accuracy	$ACC = \frac{TP}{TP+FP} = \frac{1}{N} \sum_{i=0}^{N-1} \hat{\delta}\left(\hat{\mathbf{y}}_i - \mathbf{y}_i\right)$
Precision by label	$PPV(\ell) = \frac{TP}{TP+FP} = \frac{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i-\ell) \cdot \hat{\delta}(\mathbf{y}_i-\ell)}{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i-\ell)}$
Recall by label	$TPR(\ell) = \frac{TP}{P} = \frac{\sum_{i=0}^{N-1} \hat{\delta}(\hat{\mathbf{y}}_i-\ell) \cdot \hat{\delta}(\mathbf{y}_i-\ell)}{\sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i-\ell)}$
F-measure by label	$F(\beta, \ell) = \left(1+\beta^2\right) \cdot \left(\frac{PPV(\ell) \cdot TPR(\ell)}{\beta^2 \cdot PPV(\ell)+TPR(\ell)}\right)$
Weighted precision	$PPV_w = \frac{1}{N} \sum_{\ell \in L} PPV(\ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$
Weighted recall	$TPR_w = \frac{1}{N} \sum_{\ell \in L} TPR(\ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$
Weighted F-measure	$F_w(\beta) = \frac{1}{N} \sum_{\ell \in L} F(\beta, \ell) \cdot \sum_{i=0}^{N-1} \hat{\delta}(\mathbf{y}_i - \ell)$

Examples

Scala | Java | Python

The following code snippets illustrate how to load a sample dataset, train a multiclass classification algorithm on the data, and evaluate the performance of the algorithm by several multiclass classification evaluation metrics.
Refer to the `MulticlassMetrics` Python docs for more details on the API.

```
from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.util import MLUtils
from pyspark.mllib.evaluation import MulticlassMetrics

# Load training data in LIBSVM format
data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_multiclass_classification_data.txt")

# Split data into training (60%) and test (40%)
training, test = data.randomSplit([0.6, 0.4], seed=11)
training.cache()

# Run training algorithm to build the model
model = LogisticRegressionWithLBFGS.train(training, numClasses=3)
```

```
# Compute raw scores on the test set
predictionAndLabels = test.map(lambda lp: (float(model.predict(lp.features)), lp.label))

# Instantiate metrics object
metrics = MulticlassMetrics(predictionAndLabels)

# Overall statistics
precision = metrics.precision()
recall = metrics.recall()
f1Score = metrics.fMeasure()
print("Summary Stats")
print("Precision = %s" % precision)
print("Recall = %s" % recall)
print("F1 Score = %s" % f1Score)

# Statistics by class
labels = data.map(lambda lp: lp.label).distinct().collect()
for label in sorted(labels):
    print("Class %s precision = %s" % (label, metrics.precision(label)))
    print("Class %s recall = %s" % (label, metrics.recall(label)))
    print("Class %s F1 Measure = %s" % (label, metrics.fMeasure(label, beta=1.0)))

# Weighted stats
print("Weighted recall = %s" % metrics.weightedRecall)
print("Weighted precision = %s" % metrics.weightedPrecision)
print("Weighted F(1) Score = %s" % metrics.weightedFMeasure())
print("Weighted F(0.5) Score = %s" % metrics.weightedFMeasure(beta=0.5))
print("Weighted false positive rate = %s" % metrics.weightedFalsePositiveRate)
```

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

Multilabel classification

A multilabel classification problem involves mapping each sample in a dataset to a set of class labels. In this type of classification problem, the labels are not mutually exclusive. For example, when classifying a set of news articles into topics, a single article might be both science and politics.

Because the labels are not mutually exclusive, the predictions and true labels are now vectors of label *sets*, rather than vectors of labels. Multilabel metrics, therefore, extend the fundamental ideas of precision, recall, etc. to operations on sets. For example, a true positive for a given class now occurs when that class exists in the predicted set and it exists in the true label set, for a specific data point.

Available metrics

Here we define a set D of N documents

$$D = \left\{d_0, d_1, \ldots, d_{N-1}\right\}$$

Define $L_0, L_1, \ldots, L_{N-1}$ to be a family of label sets and $P_0, P_1, \ldots, P_{N-1}$ to be a family of prediction sets where L_i and P_i are the label set and prediction set, respectively, that correspond to document d_i.

The set of all unique labels is given by

$$L = \bigcup_{k=0}^{N-1} L_k$$

The following definition of indicator function $I_A(x)$ on a set A will be necessary

$$I_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{otherwise.} \end{cases}$$

Metric	Definition
Precision	$\frac{1}{N} \sum_{i=0}^{N-1} \frac{\lvert P_i \cap L_i \rvert}{\lvert P_i \rvert}$
Recall	$\frac{1}{N} \sum_{i=0}^{N-1} \frac{\lvert L_i \cap P_i \rvert}{\lvert L_i \rvert}$
Accuracy	$\frac{1}{N} \sum_{i=0}^{N-1} \frac{\lvert L_i \cap P_i \rvert}{\lvert L_i \rvert + \lvert P_i \rvert - \lvert L_i \cap P_i \rvert}$
Precision by label	$PPV(\ell) = \frac{TP}{TP+FP} = \frac{\sum_{i=0}^{N-1} I_{P_i}(\ell) \cdot I_{L_i}(\ell)}{\sum_{i=0}^{N-1} I_{P_i}(\ell)}$
Recall by label	$TPR(\ell) = \frac{TP}{P} = \frac{\sum_{i=0}^{N-1} I_{P_i}(\ell) \cdot I_{L_i}(\ell)}{\sum_{i=0}^{N-1} I_{L_i}(\ell)}$
F1-measure by label	$F1(\ell) = 2 \cdot \left(\frac{PPV(\ell) \cdot TPR(\ell)}{PPV(\ell) + TPR(\ell)} \right)$
Hamming Loss	$\frac{1}{N \cdot \lvert L \rvert} \sum_{i=0}^{N-1} \lvert L_i \rvert + \lvert P_i \rvert - 2 \lvert L_i \cap P_i \rvert$
Subset Accuracy	$\frac{1}{N} \sum_{i=0}^{N-1} I_{\{L_i\}}(P_i)$
F1 Measure	$\frac{1}{N} \sum_{i=0}^{N-1} 2 \frac{\lvert P_i \cap L_i \rvert}{\lvert P_i \rvert \cdot \lvert L_i \rvert}$
Micro precision	$\frac{TP}{TP+FP} = \frac{\sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert}{\sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert + \sum_{i=0}^{N-1} \lvert P_i - L_i \rvert}$
Micro recall	$\frac{TP}{TP+FN} = \frac{\sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert}{\sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert + \sum_{i=0}^{N-1} \lvert L_i - P_i \rvert}$
Micro F1 Measure	$2 \cdot \frac{TP}{2 \cdot TP + FP + FN} = 2 \cdot \frac{\sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert}{2 \cdot \sum_{i=0}^{N-1} \lvert P_i \cap L_i \rvert + \sum_{i=0}^{N-1} \lvert L_i - P_i \rvert + \sum_{i=0}^{N-1} \lvert P_i - L_i \rvert}$

Examples

The following code snippets illustrate how to evaluate the performance of a multilabel classifier. The examples use the fake prediction and label data for multilabel classification that is shown below.

Document predictions:

- doc 0 - predict 0, 1 - class 0, 2
- doc 1 - predict 0, 2 - class 0, 1
- doc 2 - predict none - class 0
- doc 3 - predict 2 - class 2
- doc 4 - predict 2, 0 - class 2, 0

- doc 5 - predict 0, 1, 2 - class 0, 1
- doc 6 - predict 1 - class 1, 2

Predicted classes:

- class 0 - doc 0, 1, 4, 5 (total 4)
- class 1 - doc 0, 5, 6 (total 3)
- class 2 - doc 1, 3, 4, 5 (total 4)

True classes:

- class 0 - doc 0, 1, 2, 4, 5 (total 5)
- class 1 - doc 1, 5, 6 (total 3)
- class 2 - doc 0, 3, 4, 6 (total 4)

Scala Java Python

Refer to the `MultilabelMetrics` Python docs for more details on the API.

```
from pyspark.mllib.evaluation import MultilabelMetrics

scoreAndLabels = sc.parallelize([
    ([0.0, 1.0], [0.0, 2.0]),
    ([0.0, 2.0], [0.0, 1.0]),
    ([], [0.0]),
    ([2.0], [2.0]),
    ([2.0, 0.0], [2.0, 0.0]),
    ([0.0, 1.0, 2.0], [0.0, 1.0]),
    ([1.0], [1.0, 2.0])])

# Instantiate metrics object
metrics = MultilabelMetrics(scoreAndLabels)

# Summary stats
print("Recall = %s" % metrics.recall())
print("Precision = %s" % metrics.precision())
print("F1 measure = %s" % metrics.f1Measure())
print("Accuracy = %s" % metrics.accuracy)

# Individual label stats
labels = scoreAndLabels.flatMap(lambda x: x[1]).distinct().collect()
for label in labels:
    print("Class %s precision = %s" % (label, metrics.precision(label)))
    print("Class %s recall = %s" % (label, metrics.recall(label)))
    print("Class %s F1 Measure = %s" % (label, metrics.f1Measure(label)))

# Micro stats
print("Micro precision = %s" % metrics.microPrecision)
print("Micro recall = %s" % metrics.microRecall)
print("Micro F1 measure = %s" % metrics.microF1Measure)

# Hamming loss
print("Hamming loss = %s" % metrics.hammingLoss)
```

```
# Subset accuracy
print("Subset accuracy = %s" % metrics.subsetAccuracy)
```

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

Ranking systems

The role of a ranking algorithm (often thought of as a recommender system) is to return to the user a set of relevant items or documents based on some training data. The definition of relevance may vary and is usually application specific. Ranking system metrics aim to quantify the effectiveness of these rankings or recommendations in various contexts. Some metrics compare a set of recommended documents to a ground truth set of relevant documents, while other metrics may incorporate numerical ratings explicitly.

Available metrics

A ranking system usually deals with a set of M users

$$U = \left\{u_0, u_1, \ldots, u_{M-1}\right\}$$

Each user (u_i) having a set of N ground truth relevant documents

$$D_i = \left\{d_0, d_1, \ldots, d_{N-1}\right\}$$

And a list of Q recommended documents, in order of decreasing relevance

$$R_i = \left[r_0, r_1, \ldots, r_{Q-1}\right]$$

The goal of the ranking system is to produce the most relevant set of documents for each user. The relevance of the sets and the effectiveness of the algorithms can be measured using the metrics listed below.

It is necessary to define a function which, provided a recommended document and a set of ground truth relevant documents, returns a relevance score for the recommended document.

$$rel_D(r) = \begin{cases} 1 & \text{if } r \in D, \\ 0 & \text{otherwise.} \end{cases}$$

Metric	Definition	Notes
Precision at k	$p(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{k}\sum_{j=0}^{\min(\lvert D\rvert,k)-1} rel_{D_i}(R_i(j))$	Precision at k is a measure of how many of the first k recommended documents are in the set of true relevant documents averaged across all users. In this metric, the order of the recommendations is not taken into account.
Mean Average Precision	$MAP = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{\lvert D_i\rvert}\sum_{j=0}^{Q-1}\frac{rel_{D_i}(R_i(j))}{j+1}$	MAP is a measure of how many of the recommended documents are in the set of true relevant documents, where the order of the recommendations is taken into account (i.e. penalty for highly relevant documents is higher).

Metric	Definition	Notes
Normalized Discounted Cumulative Gain	$NDCG(k) = \frac{1}{M}\sum_{i=0}^{M-1}\frac{1}{IDCG(D_i,k)}\sum_{j=0}^{n-1}\frac{rel_{D_i}(R_i(j))}{\ln(j+1)}$ Where $n = \min\left(\max\left(\lvert R_i\rvert, \lvert D_i\rvert\right), k\right)$ $IDCG(D,k) = \sum_{j=0}^{\min(\lvert D\rvert,k)-1}\frac{1}{\ln(j+1)}$	NDCG at k is a measure of how many of the first k recommended documents are in the set of true relevant documents averaged across all users. In contrast to precision at k, this metric takes into account the order of the recommendations (documents are assumed to be in order of decreasing relevance).

»

Examples

The following code snippets illustrate how to load a sample dataset, train an alternating least squares recommendation model on the data, and evaluate the performance of the recommender by several ranking metrics. A brief summary of the methodology is provided below.

MovieLens ratings are on a scale of 1-5:

- 5: Must see
- 4: Will enjoy
- 3: It's okay
- 2: Fairly bad
- 1: Awful

So we should not recommend a movie if the predicted rating is less than 3. To map ratings to confidence scores, we use:

- 5 -> 2.5
- 4 -> 1.5
- 3 -> 0.5
- 2 -> -0.5
- 1 -> -1.5.

This mappings means unobserved entries are generally between It's okay and Fairly bad. The semantics of 0 in this expanded world of non-positive weights are "the same as never having interacted at all."

Scala | Java | Python

Refer to the `RegressionMetrics` Python docs and `RankingMetrics` Python docs for more details on the API.

```
from pyspark.mllib.recommendation import ALS, Rating
from pyspark.mllib.evaluation import RegressionMetrics, RankingMetrics

# Read in the ratings data
lines = sc.textFile("data/mllib/sample_movielens_data.txt")

def parseLine(line):
    fields = line.split("::")
    return Rating(int(fields[0]), int(fields[1]), float(fields[2]) - 2.5)
ratings = lines.map(lambda r: parseLine(r))

# Train a model on to predict user-product ratings
model = ALS.train(ratings, 10, 10, 0.01)
```

```
# Get predicted ratings on all existing user-product pairs
testData = ratings.map(lambda p: (p.user, p.product))
predictions = model.predictAll(testData).map(lambda r: ((r.user, r.product), r.rating))

ratingsTuple = ratings.map(lambda r: ((r.user, r.product), r.rating))
scoreAndLabels = predictions.join(ratingsTuple).map(lambda tup: tup[1])

# Instantiate regression metrics to compare predicted and actual ratings
metrics = RegressionMetrics(scoreAndLabels)

# Root mean squared error
print("RMSE = %s" % metrics.rootMeanSquaredError)

# R-squared
print("R-squared = %s" % metrics.r2)
```

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

Regression model evaluation

Regression analysis is used when predicting a continuous output variable from a number of independent variables.

Available metrics

Metric	Definition
Mean Squared Error (MSE)	$MSE = \frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{N}$
Root Mean Squared Error (RMSE)	$RMSE = \sqrt{\frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{N}}$
Mean Absolute Error (MAE)	$MAE = \sum_{i=0}^{N-1} \lvert \mathbf{y}_i - \hat{\mathbf{y}}_i \rvert$
Coefficient of Determination (R^2)	$R^2 = 1 - \frac{MSE}{\text{VAR}(\mathbf{y}) \cdot (N-1)} = 1 - \frac{\sum_{i=0}^{N-1} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{\sum_{i=0}^{N-1} (\mathbf{y}_i - \bar{\mathbf{y}})^2}$
Explained Variance	$1 - \frac{\text{VAR}(\mathbf{y} - \hat{\mathbf{y}})}{\text{VAR}(\mathbf{y})}$

Examples

Scala | Java | Python

The following code snippets illustrate how to load a sample dataset, train a linear regression algorithm on the data, and evaluate the performance of the algorithm by several regression metrics.
Refer to the `RegressionMetrics` Python docs for more details on the API.

```
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD
from pyspark.mllib.evaluation import RegressionMetrics
from pyspark.mllib.linalg import DenseVector
```

```python
# Load and parse the data
def parsePoint(line):
    values = line.split()
    return LabeledPoint(float(values[0]),
                        DenseVector([float(x.split(':')[1]) for x in values[1:]]))

data = sc.textFile("data/mllib/sample_linear_regression_data.txt")
parsedData = data.map(parsePoint)

# Build the model
model = LinearRegressionWithSGD.train(parsedData)

# Get predictions
valuesAndPreds = parsedData.map(lambda p: (float(model.predict(p.features)), p.label))

# Instantiate metrics object
metrics = RegressionMetrics(valuesAndPreds)

# Squared Error
print("MSE = %s" % metrics.meanSquaredError)
print("RMSE = %s" % metrics.rootMeanSquaredError)

# R-squared
print("R-squared = %s" % metrics.r2)

# Mean absolute error
print("MAE = %s" % metrics.meanAbsoluteError)

# Explained variance
print("Explained variance = %s" % metrics.explainedVariance)
```

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