DADS 6002 / CI 7301
Big Data Analytics

Machine Learning

- A task that infers patterns and relationships between variables or among data instances in a dataset.
- Output of machine learning can be a model that captures the patterns or relationships.
- For supervised machine learning, the model described the relationships between the dependent and independent variables and can be used to predict the value of dependent variable given the values of independent variables.

Machine Learning

- For unsupervised machine learning, the model depicts the hidden patterns or structures within the dataset, e.g. data clusters, association rules.
- Features or variables of an observation can be categorical (discrete, nominal or ordinal) or numerical (discrete or continuous)
- For supervised machine learning, a given dataset must be divided into two subsets,

1. Training data is used to train a model to predict the class label or dependent variable value.
2. Testing data is used to validate or evaluate the predictive performance of a trained model.

Machine Learning

Machine Learning Applications

- Classification

  - build a model to predict the class label of a given data observation

- Regression

  - build a model to predict a numerical label or dependent variable

- Clustering

  - Split a dataset into a specified number of clusters or segments, each with instances that are more similar than others in other clusters

Machine Learning

- Anomaly Detection

  - Detect anomaly instances or outliers from given dataset

- Recommendation

  - Collaborative filtering (predict unknown ratings from the known ones then recommend products with high predicted ratings)

  - Content-based (use explicitly specified properties to find similar products and then make recommendations).

Machine Learning

- Dimension Reduction

  - Eliminate features with low predictive power or transform a dataset into new one with less number of dimensions but still maintain characteristic of the original dataset for machine learning.

Machine Learning with Spark

- Spark provides two machine learning libraries

1. MLlib (more mature)
2. Spark ML (also known as the Pipeline API)

- Both are scalable that can work on big datasets.
- MLlib utilizes RDD to store and process data in memory in multi-node cluster environment.
- Spark ML utilizes Data frames to store and process data in multi-node cluster environment (data frame is abstract data type to store structured or semi-structured dataset in columnar format)

MLlib

- It provides a high-level API for machine learning and statistical analysis (e.g. summary statistical analysis, correlations, stratified Sampling, hypothesis testing, random data generation, kernel density estimation). The API is Available in Scala, Java, Python and R.
- It can be used for common machine learning tasks such as

1. Regression and Classification e.g.

   Linear Regression, Logistics Regression, Support Vector Machine, Naïve Bayes, decision Trees, Random Forest, Gradient-boosted trees, Isotonic Regression.

MLlib

2. Feature Extraction and Transformation
   - TF-IDF
   - Word2Vec
   - Standard Scaler
   - Normalizer
   - Chi-Squared Feature Selection
   - Elementwise Product

MLlib

3. Frequent Pattern Mining
   - FP-growth
   - Association Rules
   - Prefix Span (Sequential Pattern Mining)
4. Recommendation
   - Collaborative Filtering using ALS
5. Clustering
   - K means

MLlib Data Types

- MLlib primary data abstractions are
  - Vector
  - Labeled Point
  - Rating

MLlib Vector Types

- The vector type represents an indexed collection of Double-type (double precision floating point) values with zero-based (starting from 0) index of type int (integer).
- A vector can represents an element (point) in n-dimensional space.
- MLlib supports two types of vectors : dense and sparse.

MLlib Vector Types

- DenseVector – an instance of the DenseVector class stores a double-type value at each index position.
- A dense vector is generally used if a dataset does not have too many zero values.

```
import org.apache.spark.mllib.linalg._
val  denseVector = Vectors.dense(1.0,0.0,3.0)
```

The dense method creates an instance of the DenseVector class from the values provides to it as arguments (so the vector will not be very long).

MLlib Vector Types

- A variant of the dense method takes an array of Double-type as argument and returns an instances of the DenseVector class.
- SparseVector – an instance of the SparseVector class is a sparse vector, which stores only non-zero values (efficient data type to store a large dataset with many zero values).
- An instance of the SparseVector class is backed by two arrays, one stores the indices for non-zero values and the other stores the non-zero values.

MLlib Vector Types

```
import org.apache.spark.mllib.linalg._
val sparseVector = Vectors.sparse( 10,
   Array(3,6),Array(100.0, 200.0))
```

or

```
import org.apache.spark.mllib.linalg._
val sparseVector = Vectors.sparse(10,
   Seq((3,100.0), (6,200.0)))
```

MLlib Vector Types

For PySpark,

```
denseVec = Vectors.dense([1.0, 2.0, 3.0])

sparseVec = Vectors.sparse(4, {1: 1.0, 3: 5.5})
```

MLlib LabeledPoint Types

- The LabeledPoint type represents an observation in a labeled dataset.
- It contains both the label (dependent variable) and features (independent variables) of an observation.
- The label is stored as Double-type value and the features are stored as a Vector type.

MLlib LabeledPoint Types

- The label (Double-type value) can represent both numerical and categorical values.
- For a regression algorithm, the label stores a numerical value.
- For Binary classification, a label must be either 0.0 (represents a negative label) or 1.0 (represents a positive label).
- For multi-class classification, a label store a zero-based class index of an observation (e.g. 0.0, 1.0, 2.0, …)

MLlib LabeledPoint Types

```
import org.apache.spark.mllib.linalg.Vectors
Import
    org.apache.spark.mllib.regression.LabeledPoint

val positive = LabeledPoint(1.0,
    Vectors.dense(10.0,30.0,20.0))
val negative = LabeledPoint(0.0, Vectors.sparse(3,
    Array(0,2), Array(200.0,300.0)))
```

MLlib LabeledPoint Types

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

MLlib Rating Types

- The Rating type is used with recommendation algorithms.
- Rating consists of three fields.

1. The first field is named user (used id) of type Int.
2. The second field is named product (item id) of type Int.
3. The third field is named rating of type Double.

MLlib Rating Types

```
import org.apache.spark.mllib.recommendation._

val rating = Rating(100,10,3.0)
```

(an instance of the Rating class with user id = 100, item id = 10, and the rating = 3.0 is created)

For PySpark,

```
rating = Rating(100,10,3.0)
```

Regression Algorithms

- MLlib includes many regression algorithms in the form of classes and singleton objects (objects which can be instantiated only once) such as

1. Linear Regression with SGD

- SGD stands for stochastic gradient descent (used for learning the linear equation coefficients with large training dataset)

Regression Algorithms

Regression Linear Equation

$$y = w_0 + W^T X$$

$$where$$

$$X = (x_1, x_2, ..., x_n)$$

$$W = (w_1, w_2, ..., w_n)$$

Regression Algorithms

- Minimization Objective Function

$$f(W) = \frac{1}{m}\sum_{i=1}^{m} L(W, X_i, y_i)$$

$$L(W, X_i, y_i) = Loss\ Function = \frac{1}{2}(W^T X_i - y_i)^2$$

$$m = size\ of\ \ training\ dataset$$

Regression Algorithms

- In linear regression overfitting occurs when the model is "too complex".
- This usually happens when there are a large number of independent variables compared to the number of observations and/or there is multi-collinearity among these variables.
- Such a model will not generalize well to new data. That is, it will perform well on training data, but poorly on test data.

Regression Algorithms

2. Penalized Regression (regularization to prevent overfitting) such as

- Ridge Regression, use L2 regularization to prevent overfitting, i.e. add sum of square coefficients on the optimization objective to suppress some coefficients with minor contribution to the model to have their coefficients close to zero.

Regression Algorithms

- Objective function with L2 regularization

$$f(W) = \frac{1}{m}\sum_{i=1}^{m} L(W, X_i, y_i) + \lambda R(W)$$

$$L(W, X_i, y_i) = Loss\ Function = \frac{1}{2}(W^T X_i - y_i)^2$$

$$m = Size\ of\ training\ dataset$$

$$\lambda = Regularization\ parameter \geq 0$$

$$R(W) = \sum_{j=1}^{n} w_j{}^2$$

n = number of variables (# of dimensions)

Regression Algorithms

- LassowithSGD – use L1 Regularization to prevent overfitting. It adds sum of absolute values of coefficients to optimization objective and hence forces some of the coefficients with minor contribution to the model to be exactly equal to 0. The objective with L1 regularization is

$$f(W) = \frac{1}{m}\sum_{i=1}^{m} L(W, X_i, y_i) + \lambda R(W)$$

$$L(W, X_i, y_i) = Loss\ Function = \frac{1}{2}(W^T X_i - y_i)^2$$

$$m = Size\ of\ training\ dataset$$

$$\lambda = Regularization\ parameter \geq 0$$

$$R(W) = \sum_{j=1}^{n} | w_j |$$ *n* = number of variables (# of dimensions)

Regression Algorithms

- ElasticNet Regression – use both L1 and L2 regularization to prevent overfitting. The objective with L1 and L2 regularization is

$$f(W) = 1/m[\sum_{i=1}^{m} L(W, Xi, yi) + \alpha\left(\lambda\sum_{j=1}^{n}|w_j|\right) + (1-\alpha)\left(\frac{\lambda\sum_{j=1}^{n} {w_j}^2}{2}\right)]$$

$L(W, X_i, y_i) = Loss\ Function = 1/2 * (W^T X_i - y_i)^2$

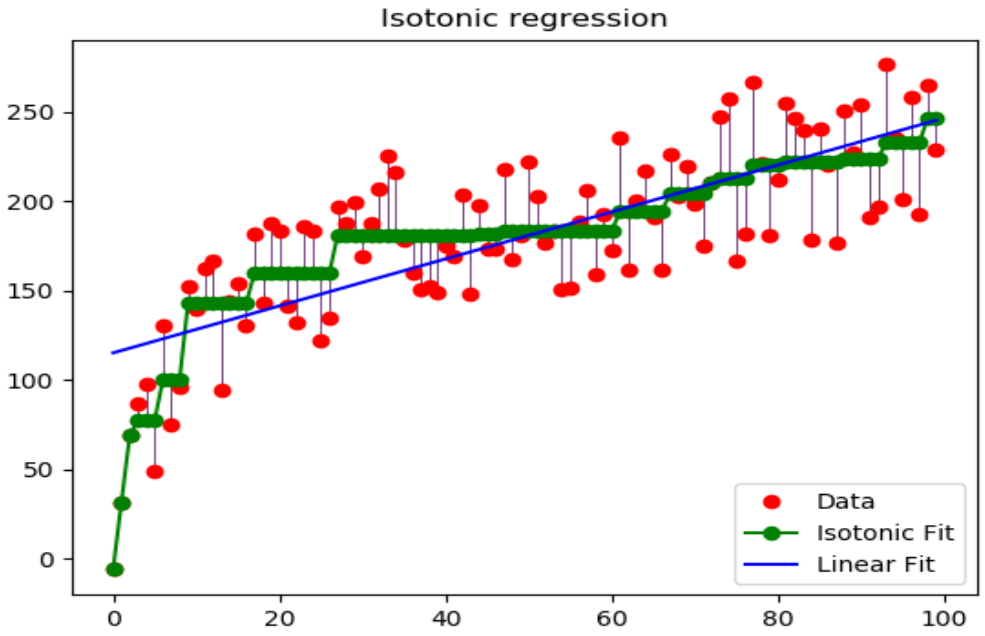
λ = Regularization Parameter >= 0

α = ElasticNet Parameter where 0 <= α <= 1

n = number of variables (# of dimensions)

Regression Algorithms

- Other regression algorithms include
- Isotonic Regression – for non linear non-decreasing function model



Source : sci-kit learn web page

Regression Algorithms

- DecisionTree (variance can be used as impurity measurement for splitting data subsets)
- GradientBoostedTrees (ensemble method using boosting)
- Random Forest (ensemble method using bagging)

Regression Algorithms

- For a linear model such as Linear Regression with SGD, Ridge Regression, *train* method can be used to build a model.
- Example

Import org.apache.spark.mllib.linalg.Vectors

Import org.apache.spark.mllib.regression.labeledPoint

Import org.apache.spark.mllib.regressionWithSGD

Regression Algorithms

```
val  lines=sc.textFile("/user/cloudera/lpsa.data")
val  labeledPoints =
        lines.map { line =>
        val Array(rawLabel,rawFeatures)
                        = line.split(',')
        val features
                = rawFeatures.split(' ').map(_.toDouble)
        LabeledPoint(rawLabel.toDouble,Vectors.dense(features))
        }
labeledPoints.cache
val  numIterations = 100
val lrModel = LinearRegressionwithSGD.train(labeledPoints, numIterations)
val intercept = lrModel.intercept
val weights = lrModel.weights
```

Regression Algorithms

- TrainRegressor – method that trains or fits a non-linear regression model such as DecisionTree and RandomForest.

val categoricalFeaturesInfo = map[Int,Int]() % list of key-value pairs, where key = id, e.g. 0, 1,2, .., of a variable which is a categorical variable, value = number of categorical values for the variable, For this example, we have an empty list means there is no categorical variables %

val Numtrees = 3

val FeatureSubsetStrategy = “auto” % Number of features to consider for splits at each node %

val impurity = “variance”

val maxDepth = 4 % max depth of the trees in the forest %

val maxBins = 32 % max number of intervals for discretization of real value variables %

Regression Algorithms

```
val rfModel = RandomForest.trainRegressor(labeledPoints,
    categoricalFeaturesInfo, NumTrees,
    FeatureSubsetStrategy, impurity, MaxDepth, maxBins)
```

Regression Algorithms

- predict – a method that predicts a numerical label for a given set of features.
- It takes an input set of features in a form of vector and returns a numerical label of type Double.

```
val observedAndPredictedLabels =
    labeledPoints.map { observation =>
    val predictedLabel = lrModel.predict(observation.features)
            (observation.label, predictedLabel)
    }
```

Classification Algorithms

- MLlib provides many classification algorithms such as

  - Linear Models such as

    LogisticRegressionWithSGD

    LogisticRegressionWithLBFGS (Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, an iterative method for solving unconstrained nonlinear optimization problems)

    SVMwithSGD

  - NaiveBayes

  - Tree-based Models (non-linear) such as

    DecisionTree

    GradientBoostedTrees

    RandomForest

Building Classification Model

- Train method of a classification object builds a classification model based on given training dataset. It takes an RDD of LabeledPoint as an argument and returns an algorithm-specific classification model.

```
val Array(trainingData,validationData,testData)=
      labeledPoints.randomSplit(Array(0.6,0.2,0.2))
trainingData.cache()
val numIterations = 100
val svmModel =
      SVMwithSGD.train(trainingData,numIterations)
```

Building Classification Model

- The trainClassifier method builds a classification models based on tree-based algorithms such as DecisionTree and RandomForest.
- It takes an RDD of LabelPoints and algorithm-specific hyperparameters as arguments.
- It returns an algorithm-specific model.

Building Classification Model

- predict method returns a class or categorical label of type Double for a given set of features in Vector format

```
val predictedLabels = testData.map { observation =>
    val predictedLabel =
    svmModel.predict(observation.features)
    (predictedLabel, observation.label)
    }
```

Building Classification Model

- The save method persists a trained model to a disk. It takes a SparkContext and path as arguments and saves the source model to the given path. The saved Model can be later read with the load method.

  svmModel.save(sc, “models/svm-model”)

- The load method is defined in the companion model objects that can be imported.

  import org.apache.spark.mllib.classification.SVMModel

  val savedSVMModel = svmModel.load(sc, “models/svm-model”)

Clustering

- MLlib provides many clustering algorithms such as

  - Kmeans (a parallelized implementation of the k-means++ method)

  - StreamingKmeans (used for clustering streaming data that arrive in batch)

  - GuassianMixture (can cluster data with elliptical shapes and sizes using probabilistic models)

  - LDA (Topic Modelling used for clustering document data)

  - Power Iteration Clustering (used for clustering graph data)

Clustering

- The train method is provided by the clustering-related singleton objects. It takes an RDD of Vector and clustering algorithm-specific parameters as arguments and returns an algorithm specific model.
- The parameters and the type of returned model depend on the clustering algorithm.
- Example – the parameters accepted by the train method for Kmeans include the number of clusters, maximum number of iterations in each run, number of parallel runs, initialization modes and random seed values for initialization.

Clustering

```
import org.apache.spark.mllib.clustering.Kmeans
import org.apache.spark.mllib.linalg.Vectors
val lines = sc.textFile(“data/mllib/Kmeans_data.txt”)
val arraysofDoubles = lines.map { line => line.split(‘
    ‘).map(_.toDouble)}
val vectors = arraysofDoubles.map { a => Vectors.dense(a)
    }.cache()
val numClusters = 2
val numIterations = 20
val KmeansModel =
    Kmeans.train(vectors,numClusters,numIterations)
```

Clustering

- Run method can also be used for clustering-related objects.

```
val numClusters = 2
val numIterations = 20
val kmeans = new
    KMeans().setMaxIterations(numIterations).setK(
    numClusters)
val KMeansModel = kmeans.run(vectors)
```

Clustering

- Predict method returns a cluster index for a given observation. It takes a Vector as an argument and return a value of type Int.

```
val obs = Vectors.dense(0.0,0.0,0.0)
val clusterIndex = KMeansModel.predict(obs)
```

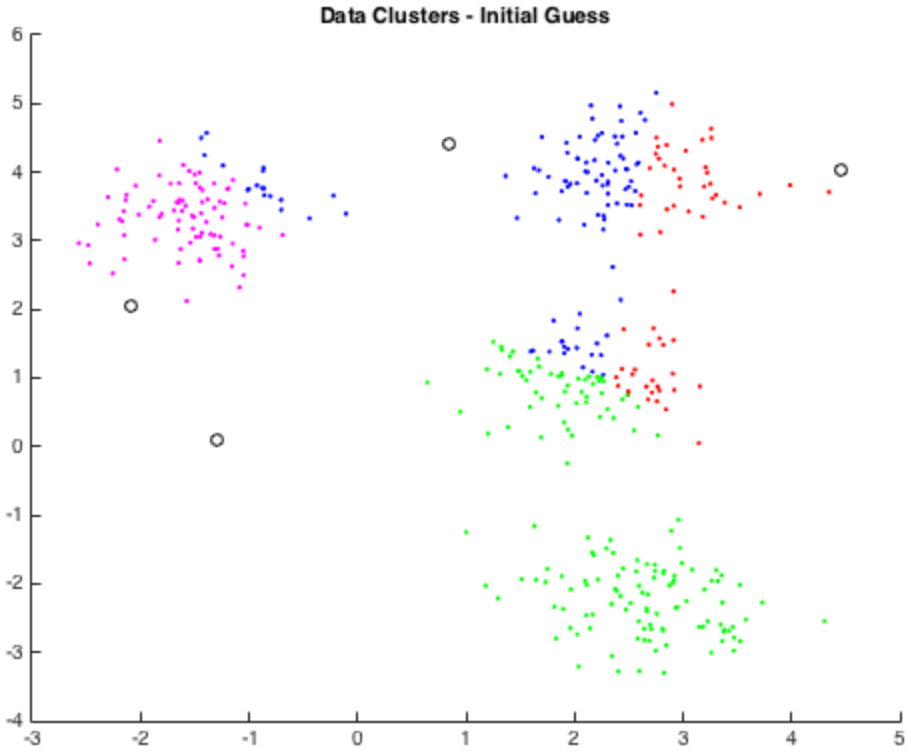
- ComputeCost method returns the sum of squared distances of the observations from their nearest cluster centers. It can be used to evaluate a KMeans model.

```
val WSSSE = KMeansModel.computeCost(vectors)
```

Clustering

- Save method persists a trained clustering model to disk. It takes a SparkContext and path as arguments and saves the source model to the given path. The saved model can be later read with the load method.

```
KMeansModel.save(sc,”models/kmean”)

import org.apache.spark.mllib.clustering.KmeansModel
val savedKmeansModel = KMeansModel.load(sc,”
                                        models/kmean”)
```



K-Means Algorithm

Collaborative Filtering

- It produces recommendations based on past behavior, preferences or similarities to known entities/users.

	item1	item2	item3	…
user1	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$	…
user2	$r_{2,1}$	$r_{2,2}$	$r_{2,3}$	…
…..	…	…	…	…

User-Item Matrix

Where $r_{i,j}$ = preference score or rating of user i on item j (e.g. 1 to 5 scale)

Collaborative Filtering

- Most of entries in the user-item matrix are unknown. Collaborative filtering tries to predict the unknown $r_{i,j}$ for user i on item j.
- Once the predicted value is computed, item j will be suggested to user i based on the predicted value (e.g. suggested if the rating is high).
- For explicit feedback, the known $r_{i,j}$ is provided directly by the user i who may use or purchase the item j. Users are asked to give the ratings for some items to the E-commerce system.

Collaborative Filtering

- For implicit feedback, the known $r_{i,j}$ is automatically inferred by the system via monitoring the different actions of the user such as history of purchases, navigation history, time spend on some web pages, links followed by the user etc. The implicit feedback reduces the burden on users to provide their preferences but it may be less accurate.

Collaborative Filtering

- Collaborative Filtering starts by building a database in the form of user-item matrix.
- The unknown preference or rating $r_{i,j}$ can be derived using the following techniques:

1. Memory techniques, to predict the value of $r_{i,j}$, K-nearest Neighboring is used to find neighboring users of user i or neighboring items of item j then the average preference or rating of the neighbors is computed to be the predicted value of $r_{i,j}$

Collaborative Filtering

- Two ways of computing $r_{i,j}$

a) User-based Collaborative filtering calculates similarity values between users based on ratings on items to find K-nearest users, then weighted average of rating based on the similarity values are computed for $r_{i,j}$

Collaborative Filtering

- The predicted value can be computed by taking a weighted average of ratings among k neighbors as follow:

$$r_{i,j} = r_i + \left(\sum_{u=1}^{k} s(u,i) * (r_{u,j} - r_u)\right) \Big/ \sum_{u=1}^{k} s(u,i))$$

where

r_i is the average ratings given by user i

r_u is the average ratings given by user u

s(u,i) is the similarity value between user u and i

Collaborative Filtering

Two most popular similarity measures are

- Correlation-based (Statistical Approach) e.g. Pearson's correlation coefficient

$$r = \frac{\sum_{i=1}^{N} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{N} (x_i - \overline{x})^2} \sqrt{\sum_{i=1}^{N} (y_i - \overline{y})^2}}$$

Collaborative Filtering

- Cosine Similarity, it is based on vector space model which views each data instance as a vector. It measures the similarity between two data instances based on the cosine value of the angle between the two vectors that represent the two data instances.
- If d_1 and d_2 are two user vectors, then

$$\cos(d_1, d_2) = (d_1 \bullet d_2) / \|d_1\| \|d_2\| ,$$

where $\bullet$ indicates vector dot product and $\| d \|$ is the length of vector d.

Collaborative Filtering Techniques

b) Item-based Collaborative filtering, it computes $r_{i,j}$ using similarity values between items instead of similarity values between users. The predicted value can be computed by taking a weighted average of ratings among k item neighbors.

- Item based collaborative filtering is a common choice for high load services (large numbers of users and items). The reason is that usually item's neighbourhood changes much slower in compared to user's neighbourhood.

Collaborative Filtering Techniques

2. Model based Collaborative Filtering

- MLlib focuses on user based recommendations using Matrix Completion techniques. The most popular technique is Alternating Least Square (ALS).
- Let M be a user-item matrix of size m x n.
- Most of entries in M are unknown. The task of matrix completion is to estimate the entries of the matrix.

Collaborative Filtering Techniques

$$M = UV^T$$

where

U is the matrix of m x k

V is the matrix of n x k

k is the rank of the model << min(m,n)

Collaborative Filtering Techniques

- The objective of ALS is to minimize the loss function

$$RMSE = \sqrt{(\sum_{i,j} (p_{i,j} - r_{i,j})^2 / N}$$

Where

- $p_{i,j}$ is the predicted rating of user i on item j
- $r_{i,j}$ is the actual rating of user i on item j
- N is the number of known ratings

Collaborative Filtering Techniques

- ALS algorithm

Step 1 : Initialize matrix V by assigning the average rating for the first row, and small random numbers for the remaining entries.

Step 2 : Fix V, solve U by minimizing the RMSE function using a regularized linear least square method.

Step 3 : Fix U, solve V by minimizing the RMSE function using a regularized linear least square method.

Step 4 : Repeat Steps 2 and 3 until convergence.

Recommendation

- The train method of the ALS object trains or fits a MatrixFactorizationModel model with an RDD of Rating.
- It takes an RDD of Rating and ALS specific parameters as arguments and returns an instance of the matrixFactorizationModelClass.
- The parameters include number of latent features (ranks), number of iterations, regularization factor, level of parallelism and random seed. The last two parameters are optional.

Recommendation

```
import org.apache.spark.mllib.recommendation.ALS
Import org.apache.spark.mllib.recommendation.Rating
val lines = sc.textFile(“data/mllib/als/test.data”)
val ratings = lines.map {  line =>
val Array( user,item,rate) = line.split(‘,’)
Rating(user.toInt, item.toInt,rate.toDouble) }
val rank = 10
val numIterations = 10
Val mfModel = ALS.train(ratings, rank, numIterations, 0.01)
```

Recommendation

- The trainImplicit method can be used when only implicit user feedback for a product is available. Feedback values r_{ui} (e.g. # views of product *i* webpage for user *u*) should be normalized to the values between 0 and 1. Instead of predicting ratings, the trainImplicit wants to predict a preference p_{ui} with a confidence value of c_{ui}. Where

$$p_{ui} = \begin{cases} 1 & r_{ui} > 0 \\ 0 & r_{ui} = 0 \end{cases} \qquad c_{ui} = 1 + \alpha r_{ui}$$

  α is a parameter for converting a feedback value to a confidence value which is greater than 1.0.

  The objective function for the machine learning is

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

  λ is a regularization parameter

Recommendation

- The trainImplicit method takes a feedback RDD and ALS-specific parameters as arguments and returns an instances of the MatrixFactorizationModel Class. (see Yifan Hu, Yehuda Koren, Chris Volinsky, Collaborative Filtering for Implicit Feedback)

Recommendation

```
val rank = 10
val numIterations = 10
val alpha = 0.01    % a parameter for converting a
                    feedback value to a confidence
                    value
val lambda = 0.01  % regularized parameter

val mfModel = ALS.trainImplicit( feedback, rank,
numIterations, lambda, alpha)
```

Recommendation

- The predict method in the MatrixFactorizationModel class returns a rating for a given user and product.
- It take user id and product id, which are of type Int, and returns a rating, which of type Double.

```
val userId = 1
val prodId = 1

val predictedRating = mfModel.predict(userId, prodId)
```

Recommendation

- A variant of the predict method accepts an RDD of pairs of user id and product id and returns an RDD of Rating.

```
val  usersProducts = ratings.map { case Rating(user, product, rate) => (user, product) }
val predictions = mfModel.predict(userProducts)
val firstFivePredictions = predictions.take(5)
```

Recommendation

- The recommendProducts method recommends the specified number of products for a given user.
- It takes a user id and number of products to recommend as arguments and returns an Array of Rating. Each Rating object includes the given user id, product id and a predicted rating score. The returned Array is sorted by rating score in descending order.

```
val userId = 1
val numProducts = 3
val recommendedProducts=
        mfModel.recommendProducts(userId, NumProducts)
```

Recommendation

- The recommendProductsForUsers method recommends the specified number of top products for all users.
- It takes the number of products to recommend as an argument and returns an RDD of (user id, Array of Rating(user id, recommended product id, rating score)).

```
val numProducts = 2
val recommendedProductsForAllUsers =
        mfModel.recommendProductsForUsers(numProducts)
```

Recommendation

- The recommendUsers method recommends the specified number of users for a given product.
- It takes the product id and the number of users to recommend as an argument and returns an Array of Rating. Each Rating object includes a user id, the given product Id and a score in the rating field.

```
val productId = 2
val numUsers = 3
val recommendedUsers =
        mfModel.recommendUsers(productId, numUsers)
```

Recommendation

- The recommendUsers method recommends the specified number of users for a given product.
- It takes the product id and the number of users to recommend as an argument and returns an Array of Rating. Each Rating object includes a user id, the given product id and a score in the rating field.

```
val productId = 2
val numUsers = 3
val recommendedUsers =
        mfModel.recommendUsers(productId, numUsers)
```

Recommendation

- The recommendUsersForProducts method recommends the specified number of users for all products.
- It takes the number of users to recommend as an argument and returns an RDD of (product id, array of Rating(product id, recommended user id, rating score)).

```
val numUsers = 2
val recommendedUsersForAllProducts =
        mfModel.recommendUsersForProducts(numUsers)
val ruFor4Products =
        recommendedUsersForAllProducts.take(4)
```

Recommendation

- The save method persists a MatrixFactorizationModel to disk. It takes a SparkContext and path of the saved file as arguments.

```
mfModel.save(sc, “models/mf-model”)
```

- The load method can be used to read a previously saved model from a file.

```
import
org.apache.spark.mllib.recommendation.MatrixFactorizatio
nModel
val savedmfModel = MatrixFactorizationModel.load(sc,
        “models/mf-model”)
```

Model Evaluation

- Regression Metrics class can be used for evaluating models generated by regression algorithms. It can calculate mean squared error, root mean squared error, mean absolute error etc.

```
val numIterations = 100
val lrModel = linearRegressionwithSGD.train(labeledPoints,
        numIterations)
val observedandPredictedLabels = labelPoints.map {
        observation =>
val predictedLabel = lrModel.predict(observation.features)
        (observation.label, predictedLabel) }
```

Model Evaluation

```
val regressionMetrics = new
        RegressionMetrics(ObservedAndPredictedLabels)
val mse = regressionMetrics.meanSquaredError
val rmse = regressionMetrics.rootMeanSquaredError
val mae = regressionMetrics.meanAbsoluteError
```

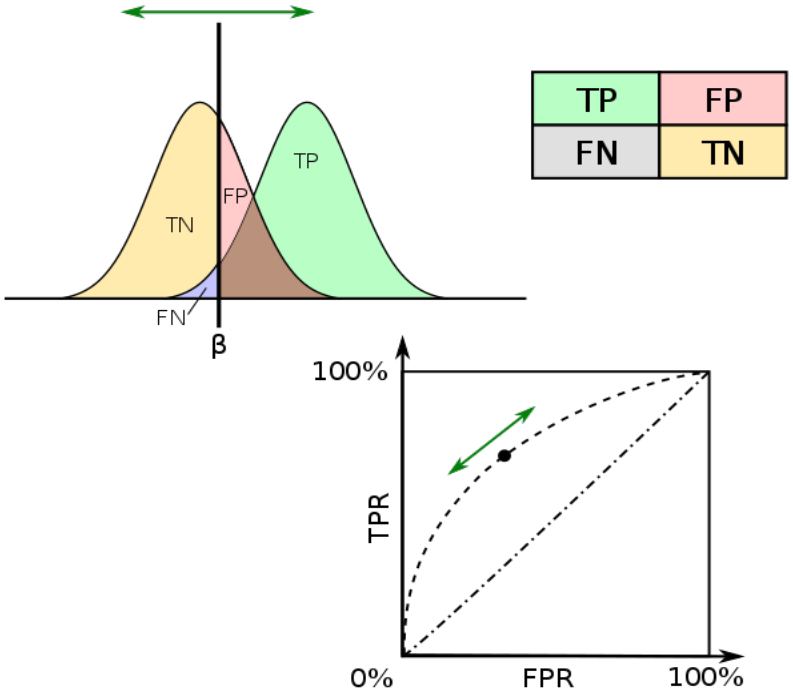
Model Evaluation

- BinaryClassificationMetrics class can be used for evaluating binary classifiers. It provides methods for calculating Receiver Operating Characteristics ROC curve (TPR vs FPR curve), area under the ROC (AUC).

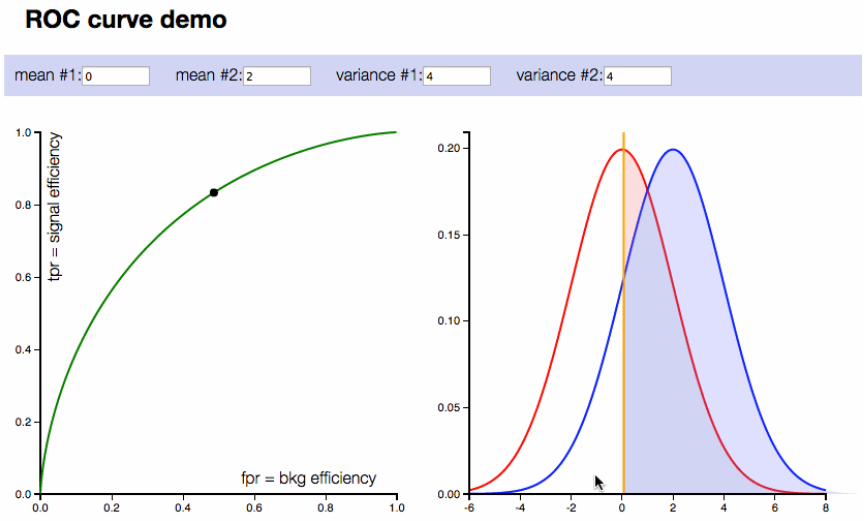
```
val svmModel = SVMWithSGD.train(trainingData,
        numIterations )
val predictedAndActualLabels = testData.map {
        observation =>
val predictedLabel =
        svmModel.predict(observation.features)
        (predictedLabel, observation.label) }
% predictedLabel is actually a score between 0 and 1.0) %
```

Model Evaluation

```
val metrics = new
        BinaryClassificationMetrics(predictedAndActualLabels)
val auROC = metrics.areaUnderROC()
```



TP	FP
FN	TN



Model Evaluation

- For evaluating a multi-class classifiers, the MulticlassMetrics class provides methods for calculating precision, recall, F-Measure etc.

```
import org.apache.spark.mllib.classification.NaiveBayes
import org.apache.spark.mllib.evaluation.MulticlassMetrics
val model = NaiveBayes.train(trainingData)
val predictionsAndLabels = testData.map { d =>
                    (model.predict(d.features), d.label) }
val metrics = new MulticlassMetrics(predictionsAndLabels)
val recall = metrics.recall
val precision = metrics.precision
val fMeasure = metrics.fMeasure
```

ML Pipelines

- ML Pipelines provide a uniform set of high-level APIs built on top of DataFrames that help users create and tune practical machine learning pipelines.
- ML Pipelines standardize APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow.

Source: ML Pipelines - Spark 2.4.0 Documentation - Apache Spark

Main Concepts in ML

- **DataFrame**: This ML API uses DataFrame from Spark SQL as an ML dataset, which can hold a variety of data types. E.g., a DataFrame could have different columns storing text, feature vectors, true labels, and predictions.
- **Transformer**: A Transformer is an algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer which transforms DataFrame with features into a DataFrame with predictions.

Main Concepts in ML

- **Estimator**: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.
- **Pipeline**: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.
- **Parameter**: All Transformers and Estimators now share a common API for specifying parameters.

Pipeline components

- **Transformers**

A Transformer is an abstraction that includes feature transformers and learned models. Technically, a Transformer implements a method transform(), which converts one DataFrame into another, generally by appending one or more columns. For example:

Pipeline components

- A feature transformer might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.
- A learning model might take a DataFrame, read the column containing feature vectors, predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.

Pipeline components

- **Estimators**

An Estimator abstracts the concept of a learning algorithm or any algorithm that fits or trains on data. Technically, an Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer. For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer.

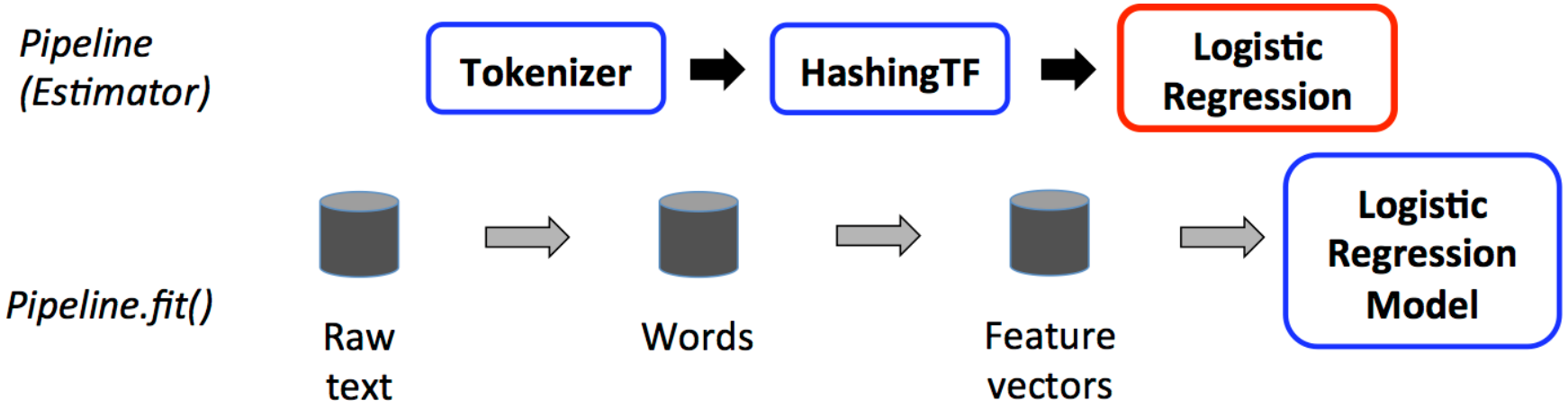
Pipeline

- In machine learning, it is common to run a sequence of algorithms to process and learn from data. E.g., a simple text document processing workflow might include several stages:
- Split each document’s text into words.
- Convert each document’s words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

Pipeline

- A Pipeline is specified as a sequence of stages, and each stage is either a Transformer or an Estimator. These stages are run in order, and the input DataFrame is transformed as it passes through each stage.
- For Transformer stages, the transform() method is called on the DataFrame.
- For Estimator stages, the fit() method is called to produce a Transformer (which becomes part of the PipelineModel, or fitted Pipeline), and that Transformer's transform() method is called on the DataFrame.

Pipeline



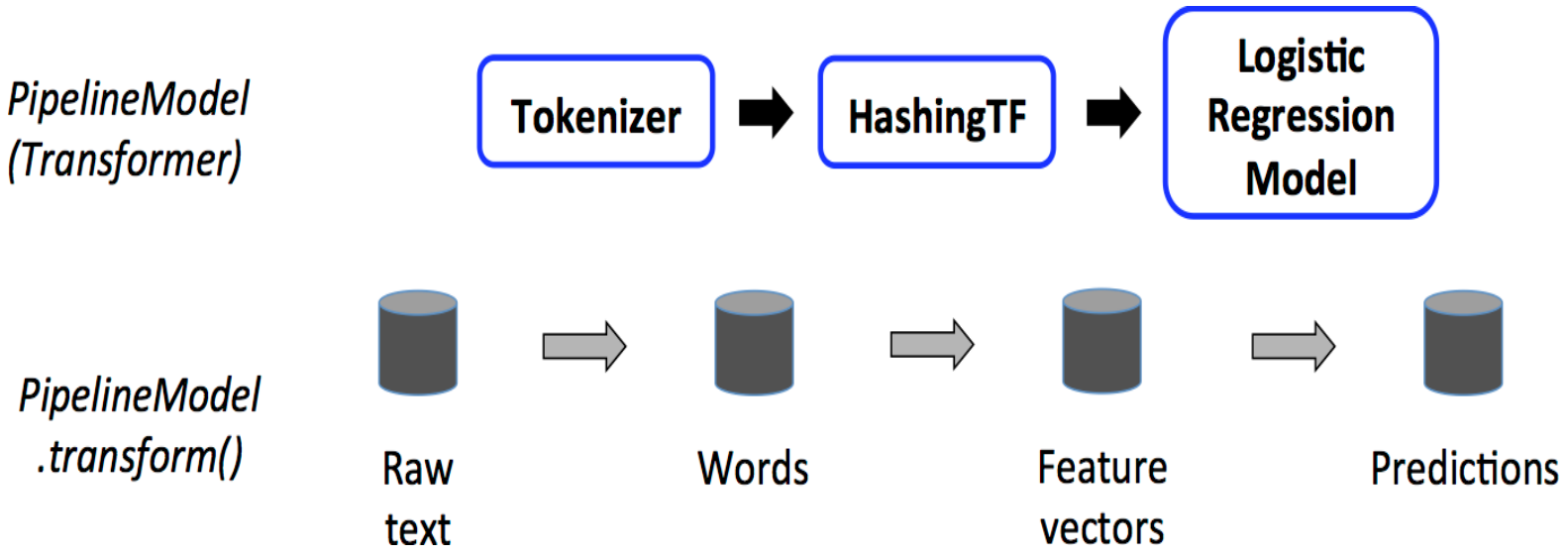
Above, the top row represents a Pipeline with three stages. The first two (Tokenizer and HashingTF) are Transformers (blue), and the third (LogisticRegression) is an Estimator (red). The bottom row represents data flowing through the pipeline, where cylinders indicate DataFrames.

Pipeline

- The Pipeline.fit() method is called on the original DataFrame, which has raw text documents and labels. The Tokenizer.transform() method splits the raw text documents into words, adding a new column with words to the DataFrame.
- The HashingTF.transform() method converts the words column into feature vectors, adding a new column with those vectors to the DataFrame.
- Now, since LogisticRegression is an Estimator, the Pipeline first calls LogisticRegression.fit() to produce a LogisticRegressionModel.

Pipeline

- A Pipeline is an Estimator. Thus, after a Pipeline's fit() method runs, it produces a PipelineModel, which is aTransformer. This PipelineModel is executed at *test time*; the figure below illustrates this usage.



Pipeline

- Estimators and Transformers use a uniform API for specifying parameters.
- A Param is a named parameter with self-contained documentation. A ParamMap is a set of (parameter, value) pairs.
- There are two main ways to pass parameters to an algorithm:

Pipeline

1. Set parameters for an instance. E.g., if lr is an instance of LogisticRegression, one could call lr.setMaxIter(10) to make lr.fit() use at most 10 iterations. This API resembles the API used in spark.mllib package.
2. Pass a ParamMap to fit() or transform(). Any parameters in the ParamMap will override parameters previously specified via setter methods.

Pipeline

- For example, if we have
  two LogisticRegressioninstances lr1 and lr2, then
  we can build a ParamMap with
  both maxIter parameters specified:

  params = ParamMap(lr1.maxIter -> 10,

  lr2.maxIter -> 20)

- This is useful if there are two algorithms with
  the maxIter parameter in a Pipeline.

  val model = pipline.fit(training, params)

ML example

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row

// Prepare training data from a list of (label, features) tuples.
val training = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(0.0, 1.1, 0.1)),
  (0.0, Vectors.dense(2.0, 1.0, -1.0)),
  (0.0, Vectors.dense(2.0, 1.3, 1.0)),
  (1.0, Vectors.dense(0.0, 1.2, -0.5))
)).toDF("label", "features")
```

ML example

```
// Create a LogisticRegression instance. This instance is an Estimator.
val lr = new LogisticRegression()

// We may set parameters using setter methods.
lr.setMaxIter(10).setRegParam(0.01)
```

ML example

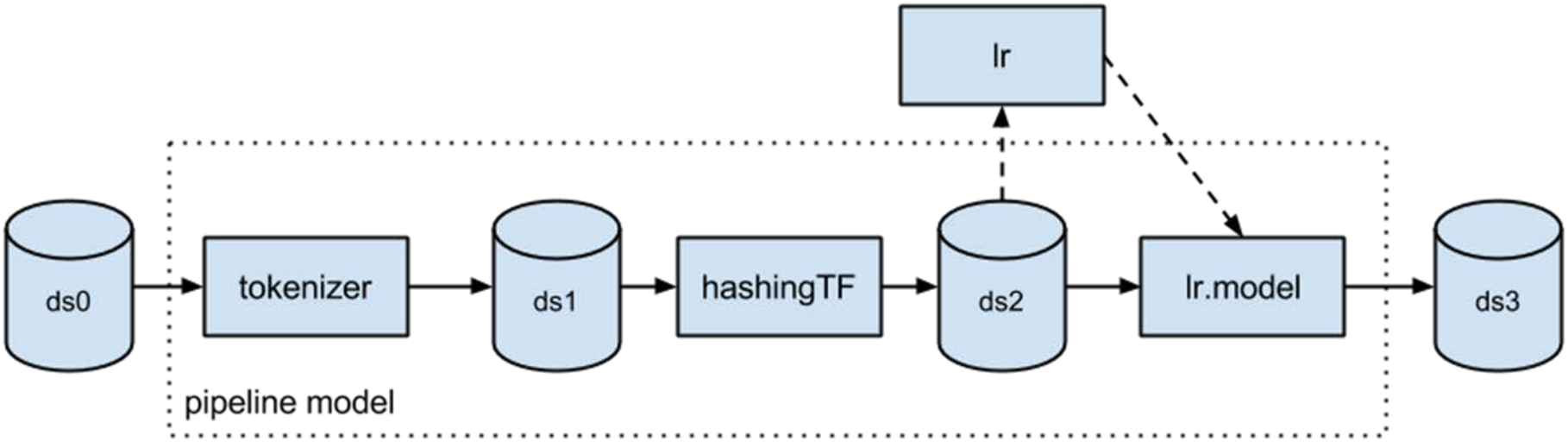
```
// Learn a LogisticRegression model. This uses the parame
ters stored in lr.
val model1 = lr.fit(training)

// Prepare test data.
val test = spark.createDataFrame(Seq(
  (1.0, Vectors.dense(-1.0, 1.5, 1.3)),
  (0.0, Vectors.dense(3.0, 2.0, -0.1)),
  (1.0, Vectors.dense(0.0, 2.2, -1.5))
)).toDF("label", "features")
```

ML example

// Make predictions on test data using the Transformer.transform() method.

```
model1.transform(test).collect()
```

ML Pipeline Example



ML Pipeline Example

```
val tokenizer = new Tokenizer()
                        .setInputCol(“text”)
                        .setOutputCol(“words”)
val hashingTF = new HashingTF()
                        .setNumFeatures(1000)
                        .setInputCol(tokenizer.getOutputCol)
                        .setOutputCol(“features”)
val lr = new LogisticRegression()
                .setMaxIter(10)
                .setRegParam(0.01)
val pipeline = new Pipeline()
                .setStages(Array(tokenizer, hashingTF, lr))
```

ML Pipeline Example

```
val model = pipeline.fit(trainingDataset)
model.transform(testDataset)
        .select(‘text, ‘label, ‘prediction)
        .collect()
        .foreach(println)
```