Grid Dynamics

Spark ML course

About us

Vitalii Monastyrev

Senior BigData engineer in Grid Dynamics, extensive experience in working with Spark, Hadoop, cloud solutions (Azure) etc.

Igor Samsonov

Senior BigData engineer in Grid Dynamics, working with Spark, Hadoop, AWS etc.

Links

Telegram link:

https://t.me/+iLug4AwtC9oyMDUy

GitHub link:

https://github.com/griddynamics/SparkMLWorkshop

Agenda

- About Spark
- Classification
- Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Custom Transformer
- Custom Estimator

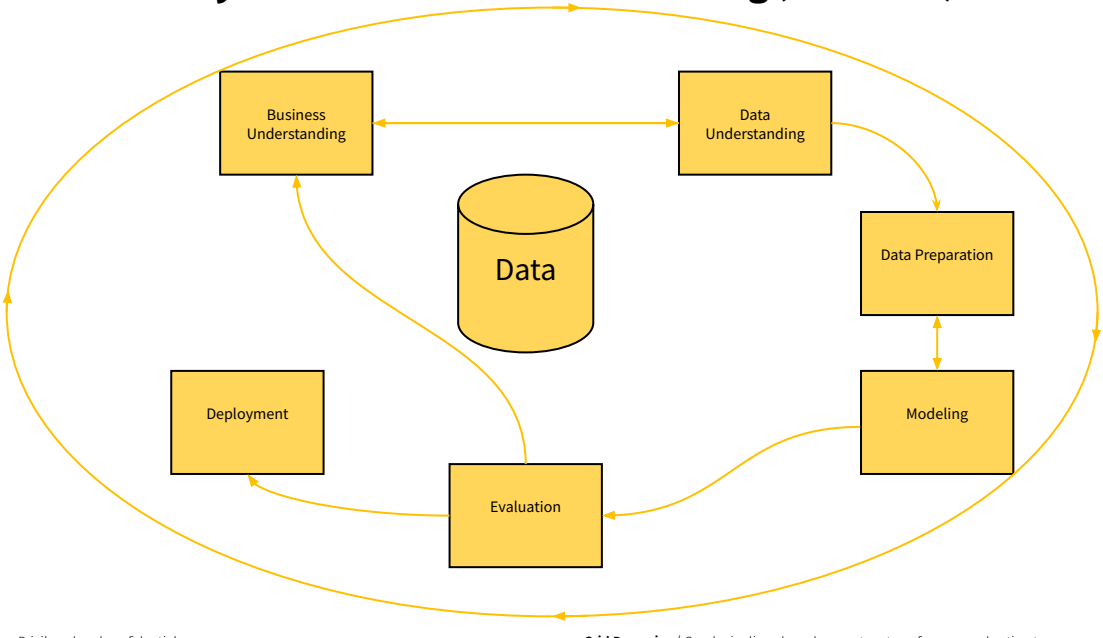
Grid Dynamics

Lecture 1: Classification

Lecture 1: Classification

- Introduction
- About Spark
- Spark MLlib
- Classification
  - Naive Bayes Classifier
  - Logistic Regression
  - Decision Tree Classifier
  - Random Forest Classifier
  - GBT (Gradient-Boosted Trees) Classifier
  - Multilayer Perceptron Classifier
- Regression
  - Linear Regression
- Clustering
  - K-Means
- Collaborative filtering
  - ALS (alternating least squares)
- Frequent Pattern Mining
  - FPG (Frequent Pattern-Growth)
- Custom Transformer
- Custom Estimator
- Summary
- QA

Cross-Industry Standard Process for Data Mining (CRISP-DM)



The process of developing a machine learning model (Pipeline)

The main problems during development:

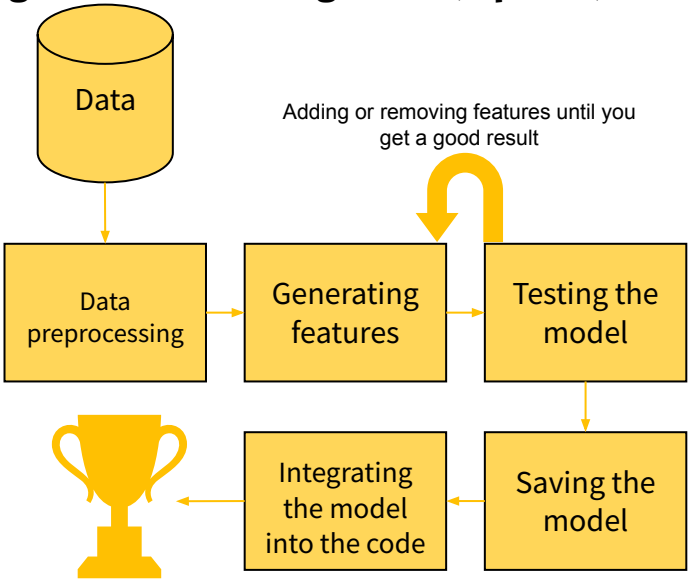
1. Building the program architecture

2. The need to quickly add and remove new features on request

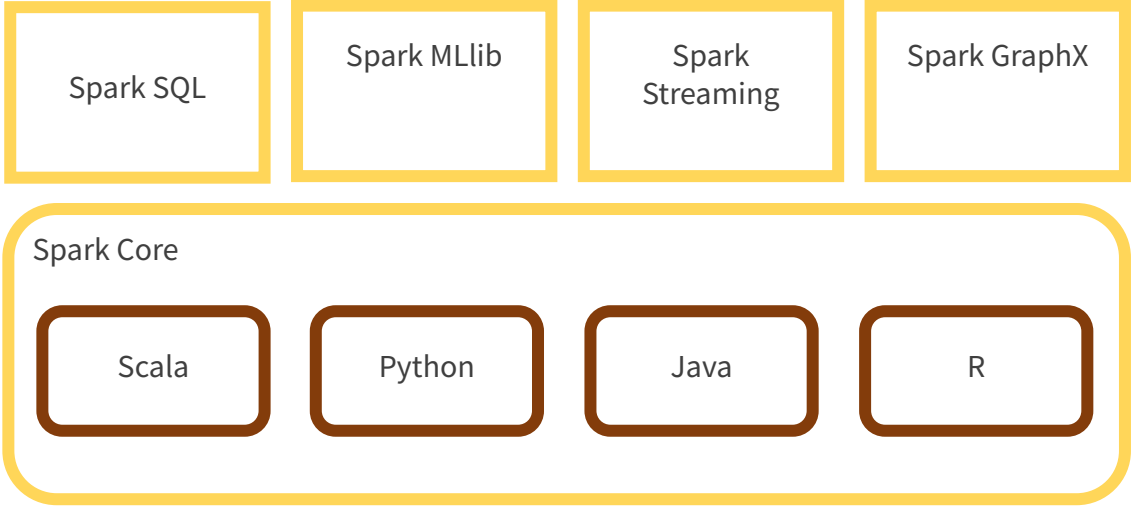
3. Integration with other programming languages (for example, Scala and Python)

4. Problems of parallelization of calculations for large amounts of data

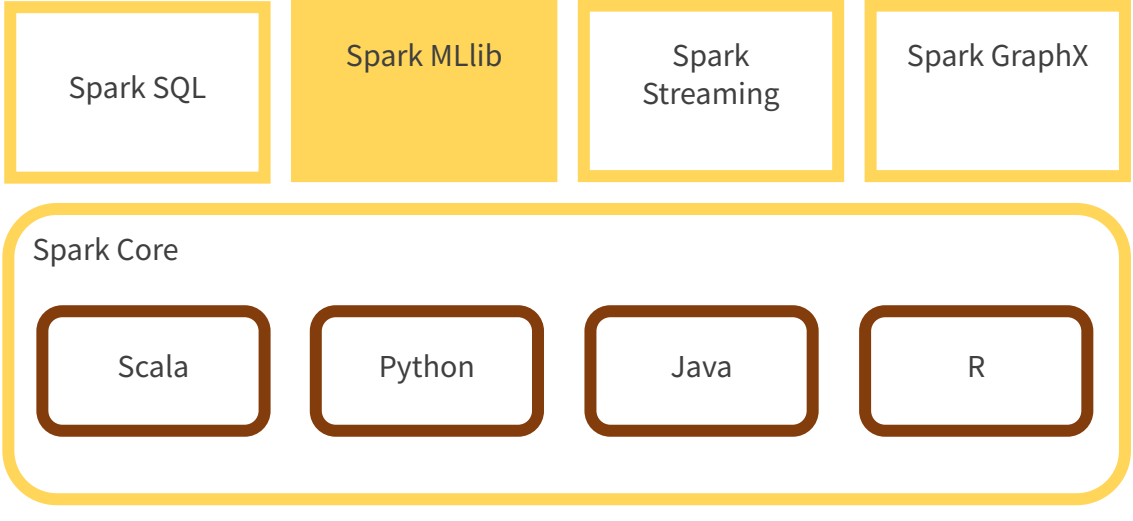
5. Integration of machine learning models from Data-science teams etc.



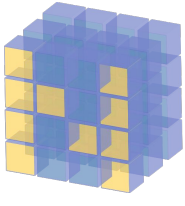
Spark


Spark SQL
Spark MLlib
Spark Streaming
Spark GraphX
Spark Core
Scala
Python
Java
R

Spark


Spark SQL
Spark MLlib
Spark Streaming
Spark GraphX
Spark Core
Scala
Python
Java
R

Machine learning libraries



NumPy

TensorFlow

Spark MLlib - standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline, or workflow

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.

It is possible to create the necessary data processing modules that will be fully integrated with Apache Spark

When pipelines are created, no additional frameworks are needed, everything works out of the box

Benefits of Spark MLlib

Simplicity

Simple APIs familiar to data scientists coming from tools like R and Python.

Scalability

Ability to run the same ML code on your laptop and on a big cluster seamlessly without breaking down.

Streamlined end-to-end

Building MLlib on top of Spark makes it possible to tackle distinct needs with a single tool instead of many disjointed ones.

Classic problems solved with machine learning

- Classification
- Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining

Tokenizer and HashingTF

Tokenization is the process of taking text (such as a sentence) and breaking it into individual terms (usually words). A simple Tokenizer class provides this functionality. The example below shows how to split sentences into sequences of words.

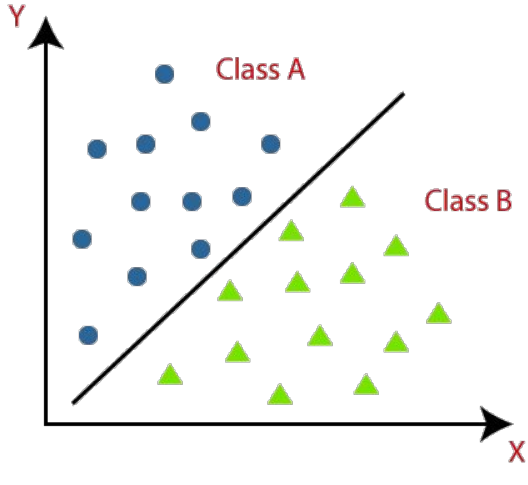
HashingTF is a Transformer which takes sets of terms and converts those sets into fixed-length feature vectors. In text processing, a “set of terms” might be a bag of words. HashingTF utilizes the hashing trick. A raw feature is mapped into an index (term) by applying a hash function.

```
+-----------------------------------------+-----------------------------------------------------------------------+
|in                                       |out                                                                    |
+-----------------------------------------+-----------------------------------------------------------------------+
|money bingo                              |[money, bingo]                                                         |
|Good afternoon, we are waiting for your answer.|[good, afternoon,, we, are, waiting, for, your, answer.]|
|money money                              |[money, money]                                                         |
+-----------------------------------------+-----------------------------------------------------------------------+
```

```
+-----------------------------------------+-----------------------------------------+
|out                                      |features                                 |
+-----------------------------------------+-----------------------------------------+
|[money, bingo]                           |(10,[2,4],[1.0,1.0])                     |
|[good, afternoon,, we, are, waiting, for, your, answer.]|(10,[3,5,6,8],[2.0,1.0,3.0,2.0])|
|[money, money]                           |(10,[2],[2.0])                           |
+-----------------------------------------+-----------------------------------------+
```

Classification

Classification is the problem of identifying which of a set of categories (sub-populations) an observation, (or observations) belongs to. Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.).



Naive Bayes Classifier

A Naive Bayes classifier is a probabilistic machine learning model that's used for classification task. The crux of the classifier is based on the Bayes theorem.

Using Bayes theorem, we can find the probability of A happening, given that B has occurred. Here, B is the evidence and A is the hypothesis. The assumption made here is that the predictors/features are independent. That is presence of one particular feature does not affect the other. Hence it is called naive.

Bayes Theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where A and B are events and $P(B) \neq 0$

- $P(A|B)$ is a conditional probability: the probability of event A occurring given that B is true. It is also called the posterior probability of A given B.
- $P(B|A)$ is also a conditional probability: the probability of event B occurring given that A is true. It can also be interpreted as the likelihood of A given a fixed B because $P(B|A) = L(A|B)$.
- $P(A)$ and $P(B)$ are the probabilities of observing A and B respectively without any given conditions; they are known as the marginal probability or prior probability.
- A and B must be different events.

Spark ML NaiveBayes

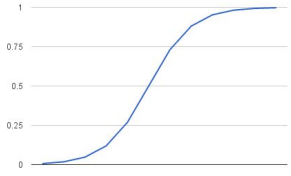
```
import org.apache.spark.ml.classification.NaiveBayes
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator

// Load the data stored in LIBSVM format as a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Split the data into training and test sets (30% held out for testing)
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3), seed = 1234L)

// Train a NaiveBayes model.
val model = new NaiveBayes()
  .fit(trainingData)

// Select example rows to display.
val predictions = model.transform(testData)
predictions.show()

// Select (prediction, true label) and compute test error
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("label")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test set accuracy = $accuracy")
```

Logistic Regression

Logistic regression is named for the function used at the core of the method, the logistic function.

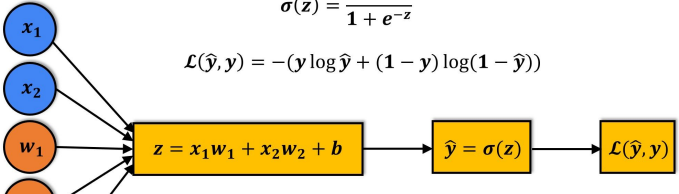
The logistic function, also called the sigmoid function was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

Logistic function



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$



where:
x1, x2 - features
w1, w2 - weights
b - parameter
L - loss function

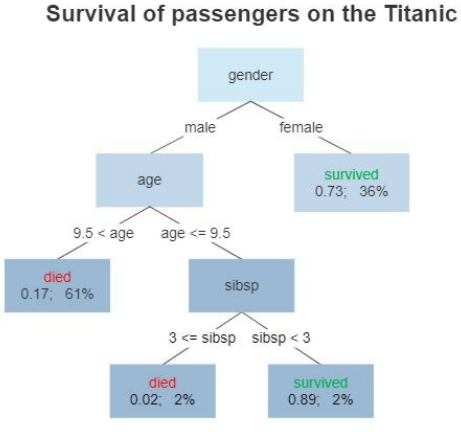
Spark ML LogisticRegression

```scala
import org.apache.spark.ml.classification.LogisticRegression

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

val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)

// Fit the model
val lrModel = lr.fit(training)

// Print the coefficients and intercept for logistic regression
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

// We can also use the multinomial family for binary classification
val mlr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)  // regularization
  .setElasticNetParam(0.8) // L1 and L2 regularization
  .setFamily("multinomial")

val mlrModel = mlr.fit(training)

// Print the coefficients and intercepts for logistic regression with multinomial family
println(s"Multinomial coefficients: ${mlrModel.coefficientMatrix}")
println(s"Multinomial intercepts: ${mlrModel.interceptVector}")
```

Decision Tree Classifier

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

The tree can be "trained" by splitting the set into subsets, based on checking the attribute values. This process, repeated recursively on each resulting subset, is called recursive partitioning. Recursion stops when a subset in a node has the same value of the target variable, or when partitioning does not add values to the predictions. This process of top-down induction of decision trees (Eng. top-down induction of decision trees, TDIDT) is an example of a greedy algorithm, and serves as the most commonly used strategy for learning decision trees from data.



Spark ML DecisionTreeClassifier

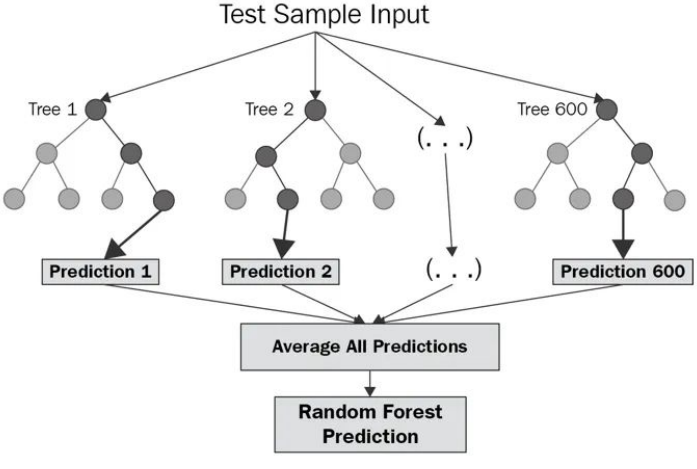
```
import org.apache.spark.ml.Pipeline
import
org.apache.spark.ml.classification.DecisionTreeClassificationModel
import org.apache.spark.ml.classification.DecisionTreeClassifier
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer,
VectorIndexer}

// Load the data stored in LIBSVM format as a DataFrame.
val data =
spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// Automatically identify categorical features, and index them.
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4) // features with > 4 distinct values are
treated as continuous.
  .fit(data)

// Split the data into training and test sets (30% held out for
testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a DecisionTree model.
val dt = new DecisionTreeClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labelsArray(0))
```

```
// Chain indexers and tree in a Pipeline.
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, dt,
labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)

// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${(1.0 - accuracy)}")

val treeModel =
model.stages(2).asInstanceOf[DecisionTreeClassificationModel]
println(s"Learned classification tree model:\n
${treeModel.toDebugString}")
```

Random Forest Classifier

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction.

The fundamental concept behind random forest is a simple but powerful one — the wisdom of crowds. In data science speak, the reason that the random forest model works so well is: A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.



Spark ML RandomForestClassifier

```
import org.apache.spark.ml.Pipeline
import
org.apache.spark.ml.classification.{RandomForestClassificationModel,
RandomForestClassifier}
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer,
VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data =
spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// Automatically identify categorical features, and index them.
// Set maxCategories so features with > 4 distinct values are
treated as continuous.
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)

// Split the data into training and test sets (30% held out for
testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a RandomForest model.
val rf = new RandomForestClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
    .setNumTrees(10)
```

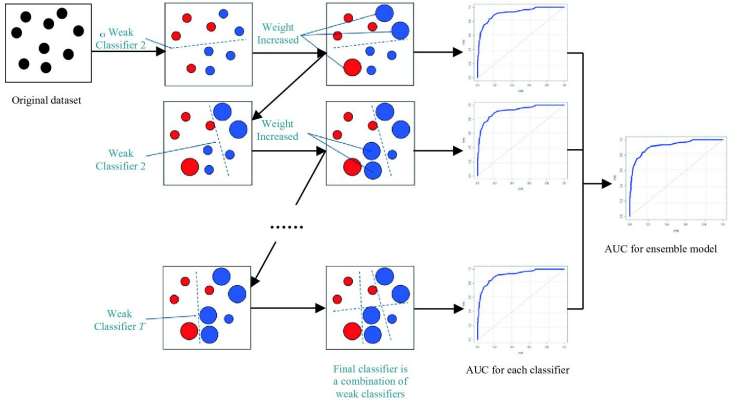
```
// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labelsArray(0))

// Chain indexers and forest in a Pipeline.
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, rf,
labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)

// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label",
"features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${(1.0 - accuracy)}")

val rfModel =
model.stages(2).asInstanceOf[RandomForestClassificationModel]
println(s"Learned classification forest model:\n
${rfModel.toDebugString}")
```

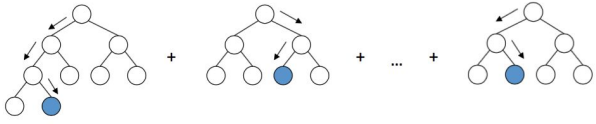
GBT (Gradient-Boosted Trees) Classifier

Gradient boosting on decision trees is a form of machine learning that works by progressively training more complex models to maximize the accuracy of predictions. Gradient boosting is particularly useful for predictive models that analyze ordered (continuous) data and categorical data. Credit score prediction which contains numerical features (age and salary) and categorical features (occupation) is one such example.

Each step of Gradient Boosting combines two steps:

1. Computing gradients of the loss function we want to optimize for each input object
2. Learning the decision tree which predicts gradients of the loss function





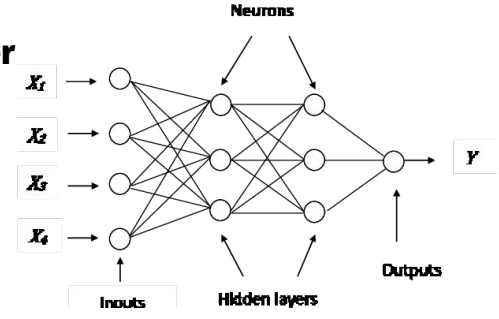
Spark ML GBTClassifier

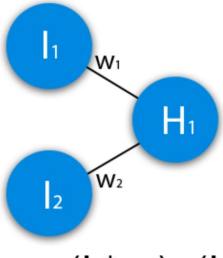
```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{GBTClassificationModel,
GBTClassifier}
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer,
VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data =
spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column.
// Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer()
  .setInputCol("label")
  .setOutputCol("indexedLabel")
  .fit(data)
// Automatically identify categorical features, and index them.
// Set maxCategories so features with > 4 distinct values are
treated as continuous.
val featureIndexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(4)
  .fit(data)

// Split the data into training and test sets (30% held out for
testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))
```

```
// Train a GBT model.
val gbt = new GBTClassifier()
  .setLabelCol("indexedLabel")
  .setFeaturesCol("indexedFeatures")
  .setMaxIter(10)
  .setFeatureSubsetStrategy("auto")

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString()
  .setInputCol("prediction")
  .setOutputCol("predictedLabel")
  .setLabels(labelIndexer.labelsArray(0))

// Chain indexers and GBT in a Pipeline.
val pipeline = new Pipeline()
  .setStages(Array(labelIndexer, featureIndexer, gbt,
labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)

// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${1.0 - accuracy}")

val gbtModel = model.stages(2).asInstanceOf[GBTClassificationModel]
   println(s"Learned classification GBT model:\n
   ${gbtModel.toDebugString}")
```

Multilayer Perceptron Classifier

Multi layer perceptron (MLP) is a supplement of feed forward neural network. It consists of three types of layers—the input layer, output layer and hidden layer. The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP. Similar to a feed forward network in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the back propagation learning algorithm. MLPs are designed to approximate any continuous function and can solve problems which are not linearly separable. The major use cases of MLP are pattern classification, recognition, prediction and approximation.





where:
I1, I2 - input neurons
w1, w2 - weights
H1 - hidden neuron

$$1) H1_{input} = (I_1 * w_1) + (I_2 * w_2)$$

$$2) H1_{output} = f_{activation}(H1_{input})$$

Spark ML MultilayerPerceptronClassifier

```
import
org.apache.spark.ml.classification.MultilayerPerceptronClassif
ier
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluat
or

// Load the data stored in LIBSVM format as a DataFrame.
val data = spark.read.format("libsvm")

.load("data/mllib/sample_multiclass_classification_data.txt
")

// Split the data into train and test
val splits = data.randomSplit(Array(0.6, 0.4), seed =
1234L)
val train = splits(0)
val test = splits(1)

// specify layers for the neural network:
// input layer of size 4 (features), two intermediate of size
5 and 4
// and output of size 3 (classes)
val layers = Array[Int](4, 5, 4, 3)
```

```
// create the trainer and set its parameters
val trainer = new
MultilayerPerceptronClassifier()
  .setLayers(layers)
  .setBlockSize(128)
  .setSeed(1234L)
  .setMaxIter(100)

// train the model
val model = trainer.fit(train)

// compute accuracy on the test set
val result = model.transform(test)
val predictionAndLabels =
result.select("prediction", "label")
val evaluator = new
MulticlassClassificationEvaluator()
  .setMetricName("accuracy")

println(s"Test set accuracy =
${evaluator.evaluate(predictionAndLabels)}"  )
```

Evaluation Metrics in ML

- Classification Metrics (accuracy, precision, recall, F1-score, ROC, AUC, ...)
- Regression Metrics (MSE, RMSE, RMSLE, MAE)
- Ranking Metrics (MRR, DCG, NDCG)
- Statistical Metrics (Correlation)
- Computer Vision Metrics (PSNR, SSIM, IoU)
- NLP Metrics (Perplexity, BLEU score)
- Deep Learning Related Metrics (Inception score, Frechet Inception distance)

Classification Evaluation Metrics

Simple example:

Classify cat images from non-cat images.

Test set: 1100 images
Non-cat images: 1000
Cat images: 100

Our model has predicted:

90 cat images correctly (true positive), 10 non correctly (false negative)

940 non-cat images correctly (true negative), 60 non correctly(false positive)

Classification Evaluation Metrics

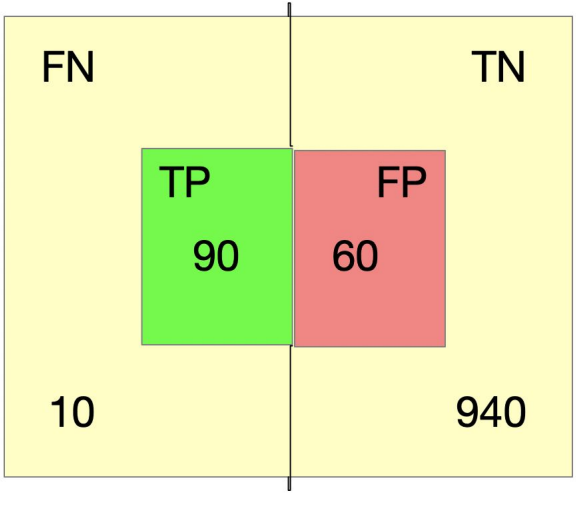
Confusion matrix is tabular visualization of the model predictions. Each row of confusion matrix represents the instances in a predicted class and each column represents the instances in actual class.

FP and FN - are confusions of classification

test set: 1100 non-cat: 1000 cat: 100		Actual	
		cat	non-cat
Predicted	cat	90 (TP)	60 (FP)
	non-cat	10 (FN)	940 (TN)

Classification Evaluation Metrics

test set: 1100 non-cat: 1000 cat: 100		Actual	
		cat	non-cat
Predicted	cat	90 (TP)	60 (FP)
	non-cat	10 (FN)	940 (TN)



Classification Evaluation Metrics

Classification accuracy: the simplest metric that can imagine, and is defined as the number of correct predictions divided by the total number of predictions, multiplied by 100. So in the above example, out of 1100 samples 1030 are predicted correctly, resulting in a classification accuracy

CA=(TP+TN)/(TP+TN+FP+FN)*100

CA=(90+940)/(90+940+60+10)=1030/1100= 93.6%

CA = (1000) / (1100) = 90,9

Example with spam:
TN = 90, FP = 10, TP = 5, FN = 5
CA= (5+90)/(5+90+10+5) = 86.4

non-spam
CA = (0 + 100)/(0+100+10+0)= 90.9

Classification Evaluation Metrics

Precision: attempts to answer the following question what proportion of positive identifications was actually correct?

Precision = TP / (TP + FP) * 100

Precision (cat) = 90 / (90 + 60) * 100 = 60%

Precision (non cat) = 940/950= 98.9%

	cat	non-cat
cat	90 (TP)	60 (FP)
non-cat	10 (FN)	940 (TN)

Classification Evaluation Metrics

Recall: is defined as the fraction of samples from a class which are correctly predicted by the model

Recall= TP/ (TP+ FN)

Recall (cat) = 90 / 100 = 90%

Recall (non cat) = 940/1000= 94%

	cat	non-cat
cat	90 (TP)	60 (FP)
non-cat	10 (FN)	940 (TN)

Classification Evaluation Metrics

F1-score is popular metric which combines precision and recall and is the harmonic mean of precision and recall

F1= 2*Precision*Recall/(Precision+Recall)

F1_cat= 2*0.6*0.9/(0.6+0.9)= 72%

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}.$$

Classification Evaluation Metrics

Sensitivity and Specificity other popular metrics mostly used in medical and biology related fields

Sensitivity= Recall= TP/(TP+FN)

Specificity= True_Negative_Rate= TN/(TN+FP)

Sensitivity = 90 / (90+10) * 100 = 90 %
Specificity = 940 / (940+60) * 100 = 94%

Task 1. Classification model. Identification of survivors based on data on passengers of the Titanic.

The task is to train a machine learning model that will predict whether the passenger survived the crash or not based on the presented data.

Conditions:

- Model should be trained only using Spark ML base classes. You can see an example of such a project in SparkMLPipeline folder.
- You need to train the model on a training data set `train.csv`.
- You need to test the model on a test dataset `test.csv`.
- Calculate metrics based on the file `is_survived.csv`.