MLlib es la biblioteca de Spark para funciones de aprendizaje automático. Diseñado para funcionar en paralelo en clústeres, MLlib contiene una variedad de algoritmos de aprendizaje y es accesible desde todos los lenguajes de programación de Spark.

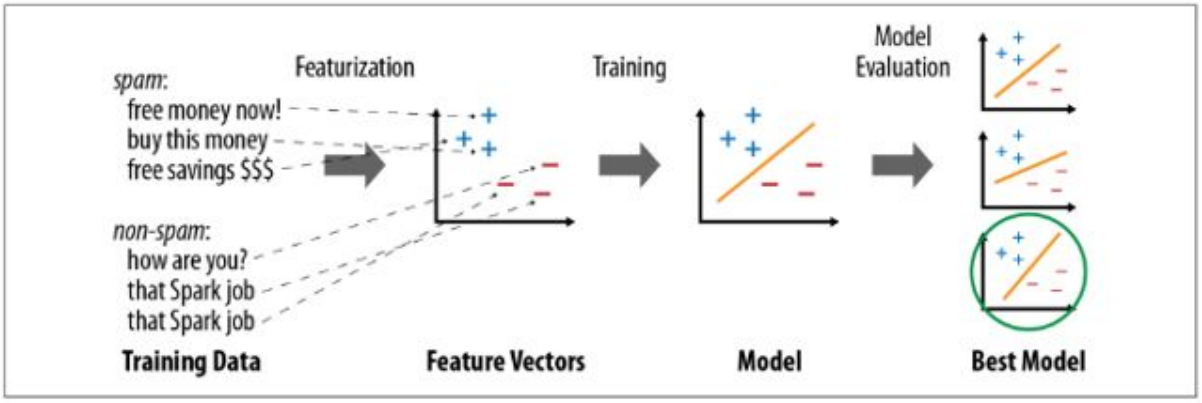
Desde el punto de vista del desarrollador permite invocar varios algoritmos en conjuntos de datos distribuidos, representando todos los datos como RDDs. Puede usar MLlib para una tarea de clasificación de texto (por ejemplo, para identificar correos spam), siguiendo los siguientes pasos:

1. Crear un RDD que represente un mensaje de correo
2. Ejecutar uno de los algoritmos de extracción de características de MLlib a partir de un texto. Esto devolverá un RDD como vector de características.
3. Utilizar a un algoritmo de clasificación (por ejemplo, regresión logística) en el RDD de vector de características;Esto devolverá un modelo que se puede utilizar para clasificar nuevos items.
4. Evaluar el modelo en un conjunto de datos de prueba utilizando una de las funciones de evaluación de MLlib.

Por ejemplo, para la clasificación de correo como spam existen varios métodos para caracterizar el texto, como contar la frecuencia de cada palabra.

Una vez que los datos se representan como vectores de características, la mayoría de los algoritmos de aprendizaje automático optimizan una función matemática bien definida basada en estos vectores. Por ejemplo, un algoritmo de clasificación puede ser definir el plano (en el espacio de vectores de características) que "mejor" separa los ejemplos de spam vs no-spam.

Al final, el algoritmo devolverá un modelo que representa la decisión de aprendizaje (por ejemplo, el plano elegido). Este modelo ahora se puede usar para hacer predicciones sobre nuevos items (por ejemplo, ver a qué lado del vector de características para un nuevo correo electrónico, para decidir si es spam).Esto se puede ver de forma gráfica en la siguiente figura:



La mayoría de los algoritmos de aprendizaje tienen múltiples parámetros que pueden afectar a los resultados, por lo que los algoritmos en el mundo real entrenarán varias versiones de un modelo y evaluarán cada una. Para ello, es común separar los datos de entrada en conjuntos de "entrenamiento" y "prueba", y entrenar sólo con el conjunto de entrenamiento, de modo que el conjunto de prueba pueda usarse para ver si el modelo confirma los datos de entrenamiento. MLlib proporciona varios algoritmos para la evaluación de modelos.

Clasificación de correo spam

Este programa utiliza dos algoritmos de MLlib: **HashingTF**, que construye los vectores de características a partir del texto teniendo en cuenta la frecuencia de términos, y **LogisticRegressionWithSGD**, que implementa el procedimiento de regresión logística usando usando el descenso de gradiente estocástico (SGD). Suponemos que comenzamos con dos archivos, spam.txt y normal.txt, cada uno de los cuales contiene ejemplos de correos spam y no spam, uno por línea.

A continuación, convierten el texto de cada archivo en un vector de características con TF y forman un modelo de regresión logística para separar los dos tipos de correos.

También usamos LabeledPoints como tipo de datos MLlib para extraer vectores de características que tengan asociados una etiqueta.

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# Cree una instancia de HashingTF para asignar texto de correo electrónico a vectores de #10.000 características.
tf = HashingTF(numFeatures = 10000)

# Cada correo electrónico se divide en palabras, y cada palabra se asigna a una característica.
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email:
tf.transform(normal.split(" ")))

# Crear conjuntos de datos LabeledPoint para ejemplos spam y no spam
spamExamples = spamFeatures.map(lambda features:
LabeledPoint(1, features))
nospamExamples = normalFeatures.map(lambda features:
LabeledPoint(0, features))

#Generamos los datos de entrenamiento
trainingData = spamExamples.union(nospamExamples)
trainingData.cache()

# Ejecutar el algoritmo de Logistic Regression a partir de los datos de entrenamiento
model = LogisticRegressionWithSGD.train(trainingData)

# Probamos con casos de spam y no spam
```

```
#Primero aplicamos la misma transformación de extracción de
vectores de características #de HashingTF, luego aplicamos el
modelo.
posTest = tf.transform("O M G GET cheap stuff by sending money
to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying Spark the other
...".split(" "))

print "Prediction for positive test example: %g" %
model.predict(posTest)
print "Prediction for negative test example: %g" %
model.predict(negTest)
```

Algoritmos MLlib

En esta sección, cubriremos los algoritmos clave disponibles en MLlib, así como sus tipos de entrada y salida.No tenemos espacio para explicar cada algoritmo matemáticamente, de esta forma nos centraremos en cómo llamar y configurar estos algoritmos.

Extracción de características

El paquete **mllib.feature** contiene varias clases para transformaciones de características comunes. Estas incluyen algoritmos para construir vectores de características a partir de un texto, y diferentes maneras de normalizar y escalar características.

El **algoritmo TF-IDF**, es uno de los más utilizados para generar vectores de características a partir de documentos de texto (por ejemplo, páginas web). Calcula dos estadísticas para cada término en cada documento: el término frecuencia (TF), que es el número de veces que aparece el término en ese documento, y la frecuencia del documento inverso (IDF), que mide con qué frecuencia ocurre un término través de todo el corpus del documento.

El producto de estos valores, TF × IDF, muestra cuánto de importante o relevante es un término para un documento específico (es decir, si es común en ese documento pero es poco común en todo el corpus).

MLlib tiene dos algoritmos que calculan TF-IDF: HashingTF y IDF, ambos en el paquete mllib.feature.

HashingTF calcula un vector de frecuencias de un término a partir de un documento.

HashingTF calcula el código de hash de cada palabra del texto modulo un tamaño de vector S, y así mapea cada palabra a un número entre 0 y S-1. Esto siempre produce un vector S-dimensional, y en la práctica es bastante robusto incluso si varias palabras se asignan al mismo código hash

HashingTF puede ejecutarse en un documento a la vez o sobre un de spark.

Requiere que cada "documento" se represente como una secuencia iterable o lista de objetos.

Ejemplo que usa HashingTF en Python.

```
>>> from pyspark.mllib.feature import HashingTF
>>> sentence = "hello world"
>>> words = sentence.split() # Divide el texto en una lista de términos
>>> tf = HashingTF(10000) # Crea un vector de tamaño S = 10,000
>>> tf.transform(words) #crea un vector de características
(10000, {3065: 1.0, 6861: 2.0})

#lo mismo se puede aplicar sobre un RDD

>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
>>> tfVectors = tf.transform(rdd) # Transforms an entire RDD
```

Una vez se han construido los vectores de frecuencia de término, se puede usar IDF para calcular las frecuencias de documento inverso y multiplicarlas por las frecuencias de término para calcular el término TF-IDF.

Primero llama a fit () para obtener un modelo IDF que representa las frecuencias del documento inverso en el corpus, luego llama a transform () en el modelo para transformar vectores TF en vectores IDF.

```
from pyspark.mllib.feature
import HashingTF, IDF

rdd = sc.wholeTextFiles("data").map(lambda (name, text):
text.split())
tf = HashingTF()
tfVectors = tf.transform(rdd).cache()

# Crear el modelo de IDF
idf = IDF()

#obtener las frecuencias del documento inverso a partir del vector
de características
idfModel = idf.fit(tfVectors)

#transformar vectores TF en vectores IDF
tfIdfVectors = idfModel.transform(tfVectors)
```