Learn Models using ML Pipeline in Spark

```
tdend2@node00:~
>>> from pyspark.ml.linalg import Vectors
ation import LogisticRegression>>> from pyspark.ml.classification import LogisticRegression
>>>
```

Created a simple training dataset using createDataFrame(). The first column has labels, e.g., 1.0 or 0.0, and the second column has dense vectors as features (vectors).

```
training = spark.createDataFrame([
(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]))], ["label", "features"])
```

```
tdend2@node00:~
>>> # Prepare training data from a list of (label, features) tuples.
... training = spark.createDataFrame([
...  (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]))], ["label", "features"])
>>>
```

The next step is setting up parameters for ML algorithms, LogisticRegression. We give 10 for maxIter (Max Iteration) and 0.01 for regParam (Regularization parameter)

```
# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(maxIter=10, regParam=0.01)
# Print out the parameters, documentation, and any default values.
print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")
```

From the below output , it is found that

maxIter: max number of iterations (>= 0). (default: 100, current: 10)
regParam: regularization parameter (>= 0). (default: 0.0, current: 0.01)
threshold: Threshold in binary classification prediction, in range [0, 1]. If threshold and thresholds are both set, they must match.e.g. if threshold is p, then thresholds must be equal to [1-p, p]. (default: 0.5)
thresholds: Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the number of classes, with values > 0, excepting that at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the class's threshold. (undefined)
tol: the convergence tolerance for iterative algorithms (>= 0). (default: 1e-06)
upperBoundsOnCoefficients: The bound matrix must be compatible with the shape (1, number of features) for binomial regression, or (number of classes, number of features) for multinomial regression. (undefined)
upperBoundsOnIntercepts:The bound vector size must be equal with 1 for binomial regression, or the number of classes for multinomial regression. (undefined)
weightCol: weight column name. If this is not set or empty, we treat all instance weights as 1.0. (undefined)

```
tdend2@node00:~
>>> # Create a LogisticRegression instance. This instance is an Estimator.
... lr = LogisticRegression(maxIter=10, regParam=0.01)
out the parameters, documentation, and any default values.
print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")>>> # Print out the parameters, documentation, and any default values.
... print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")
LogisticRegression parameters:
aggregationDepth: suggested depth for treeAggregate (>= 2). (default: 2)
elasticNetParam: the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 p
family: The name of family which is a description of the label distribution to be used in the model. Supported options: auto, binomial, mult
featuresCol: features column name. (default: features)
fitIntercept: whether to fit an intercept term. (default: True)
labelCol: label column name. (default: label)
lowerBoundsOnCoefficients: The lower bounds on coefficients if fitting under bound constrained optimization. The bound matrix must be compat
mial regression, or (number of classes, number of features) for multinomial regression. (undefined)
lowerBoundsOnIntercepts: The lower bounds on intercepts if fitting under bound constrained optimization. The bounds vector size must beequal
ses for multinomial regression. (undefined)
maxIter: max number of iterations (>= 0). (default: 100, current: 10)
predictionCol: prediction column name. (default: prediction)
probabilityCol: Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated probability estimates
es, not precise probabilities. (default: probability)
rawPredictionCol: raw prediction (a.k.a. confidence) column name. (default: rawPrediction)
regParam: regularization parameter (>= 0). (default: 0.0, current: 0.01)
standardization: whether to standardize the training features before fitting the model. (default: True)
threshold: Threshold in binary classification prediction, in range [0, 1]. If threshold and thresholds are both set, they must match.e.g. if
-p, p]. (default: 0.5)
thresholds: Thresholds in multi-class classification to adjust the probability of predicting each class. Array must have length equal to the
at most one value may be 0. The class with largest value p/t is predicted, where p is the original probability of that class and t is the cl
tol: the convergence tolerance for iterative algorithms (>= 0). (default: 1e-06)
upperBoundsOnCoefficients: The upper bounds on coefficients if fitting under bound constrained optimization. The bound matrix must be compat
mial regression, or (number of classes, number of features) for multinomial regression. (undefined)
upperBoundsOnIntercepts: The upper bounds on intercepts if fitting under bound constrained optimization. The bound vector size must be equal
asses for multinomial regression. (undefined)
weightCol: weight column name. If this is not set or empty, we treat all instance weights as 1.0. (undefined)

>>>
```

Learn model

It's recommended to use the native math library in the production system.

```
tdend2@node00:~
>> # Learn a LogisticRegression model. This uses the parameters stored in lr.
.. model1 = lr.fit(training)
4/11/04 11:25:13 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
4/11/04 11:25:13 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
>>
```

Trained model, model1

```
>>> print("Model 1 was fit using parameters: ")
Model 1 was fit using parameters:
>>> print(model1.extractParamMap())
{Param(parent=u'LogisticRegression_5994bc446e0d', name='regParam', doc='regularization parameter
f family which is a description of the label distribution to be used in the model. Supported opt
ame='labelCol', doc='label column name'): 'label', Param(parent=u'LogisticRegression_5994bc446e0
LogisticRegression_5994bc446e0d', name='rawPredictionCol', doc='raw prediction (a.k.a. confidenc
ercept', doc='whether to fit an intercept term'): True, Param(parent=u'LogisticRegression_5994bc
ram(parent=u'LogisticRegression_5994bc446e0d', name='predictionCol', doc='prediction column name
whether to standardize the training features before fitting the model'): True, Param(parent=u'Lo
itional probabilities. Note: Not all models output well-calibrated probability estimates! These
m(parent=u'LogisticRegression_5994bc446e0d', name='featuresCol', doc='features column name'): 'f
inary classification prediction, in range [0, 1]'): 0.5, Param(parent=u'LogisticRegression_5994b
ha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty'): 0.0, Param(parent=u'
}
>>>
```

Specify the parameters using a Python dictionary.

We specify the maxIter and overwrite it with 30. We can also use multiple parameters at once, regParam and threshold.

```
>>> paramMap = {lr.maxIter: 20}
>>> paramMap[lr.maxIter] = 30 # Specify 1 Param, overwriting the original maxIter.
>>> paramMap.update({lr.regParam: 0.1, lr.threshold: 0.55}) # Specify multiple Params.
>>>
```

paramMap2 = {lr.probabilityCol: "myProbability"} # Change output column name
paramMapCombined = paramMap.copy() paramMapCombined.update(paramMap2)

```
tdend2@node00:~
>>> paramMap2 = {lr.probabilityCol: "myProbability"} # Change output column name
>>> paramMapCombined = paramMap.copy()
>>> paramMapCombined.update(paramMap2)
>>>
```

Learn model with combined paramMaps

model2 = lr.fit(training, paramMapCombined)
print("Model 2 was fit using parameters: ")
print(model2.extractParamMap())

```
tdend2@node00:~
>>> model2 = lr.fit(training, paramMapCombined)
>>> print("Model 2 was fit using parameters: ")
Model 2 was fit using parameters:
>>> print(model2.extractParamMap())
{Param(parent=u'LogisticRegression_5994bc446e0d', name='regParam', doc='regularization parameter
 family which is a description of the label distribution to be used in the model. Supported optio
me='labelCol', doc='label column name'): 'label', Param(parent=u'LogisticRegression_5994bc446e0d'
ogisticRegression_5994bc446e0d', name='rawPredictionCol', doc='raw prediction (a.k.a. confidence)
rcept', doc='whether to fit an intercept term'): True, Param(parent=u'LogisticRegression_5994bc44
am(parent=u'LogisticRegression_5994bc446e0d', name='predictionCol', doc='prediction column name')
hether to standardize the training features before fitting the model'): True, Param(parent=u'Logi
tional probabilities. Note: Not all models output well-calibrated probability estimates! These pr
am(parent=u'LogisticRegression_5994bc446e0d', name='featuresCol', doc='features column name'): 'fe
binary classification prediction, in range [0, 1]'): 0.55, Param(parent=u'LogisticRegression_5994
lpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty'): 0.0, Param(parent=u'
30}
>>>
```

Prepare testing dataset

Now we have two models, model1 and model2. To test the models, we need to have a test dataset. We have a simple test dataset below. It has three lines

Prepare test data

test = spark.createDataFrame([

(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]))], ["label", "features"])

```
tdend2@node00:~
>>> test = spark.createDataFrame([
...  (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]))], ["label", "features"])
>>>
```

Make prediction

Make predictions using transform() with the test dataset

```
tdend2@node00:~
>>> prediction = model2.transform(test)
("features", "label", "myProbability", "prediction") \
.collect()>>> result = prediction.select("features", "label", "myProbability", "prediction") \
... .collect()
>>>
```

Show result

for row in result:

print("features=%s, label=%s -> prob=%s, prediction=%s"

% (row.features, row.label, row.myProbability, row.prediction))

```
tdend2@node00:~
>>> for row in result:
...  print("features=%s, label=%s -> prob=%s, prediction=%s"
...  % (row.features, row.label, row.myProbability, row.prediction))
...
features=[-1.0,1.5,1.3], label=1.0 -> prob=[0.0570730417103,0.94292695829], prediction=1.0
features=[3.0,2.0,-0.1], label=0.0 -> prob=[0.92385223117,0.0761477688296], prediction=0.0
features=[0.0,2.2,-1.5], label=1.0 -> prob=[0.109727761148,0.890272238852], prediction=1.0
>>>
```

Learn the ML pipeline in the link below.

https://archive.apache.org/dist/spark/docs/2.4.0/ml-pipeline.html#example-pipeline

Estimator, Transformer, and Param:

```
tdend2@node00:~
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.feature import HashingTF, Tokenizer
>>>
```

Prepare training documents from a list of (id, text, label) tuples.

```
tdend2@node00:~
>>> training = spark.createDataFrame([
...     (0, "a b c d e spark", 1.0),
...     (1, "b d", 0.0),
...     (2, "spark f g h", 1.0),
...     (3, "hadoop mapreduce", 0.0)
... ], ["id", "text", "label"])
>>>
```

Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

```
tdend2@node00:~
>>> tokenizer = Tokenizer(inputCol="text", outputCol="words")
ngTF = HashingTF(inputCol=tokenizer.getO>>> hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
on(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF,>>> lr = LogisticRegression(maxIter=10, regParam=0.001)
 lr])>>> pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
>>>
```

Fit the pipeline to training documents.

```
>>> model = pipeline.fit(training)
>>>
```

Prepare test documents, which are unlabeled (id, text) tuples.

```
tdend2@node00:~
>>> test = spark.createDataFrame([
...     (4, "spark i j k"),
...     (5, "l m n"),
...     (6, "spark hadoop spark"),
...     (7, "apache hadoop")
... ], ["id", "text"])
>>>
```

Make predictions on test documents and print columns of interest.

```
tdend2@node00:~
>>> prediction = model.transform(test)
, "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))>>> selected = prediction.select("id", "text", "probability", "prediction")
>>> for row in selected.collect():
...     rid, text, prob, prediction = row
...     print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
...
(4, spark i j k) --> prob=[0.159640773879,0.840359226121], prediction=1.000000
(5, l m n) --> prob=[0.837832568548,0.162167431452], prediction=0.000000
(6, spark hadoop spark) --> prob=[0.0692663313298,0.93073366867], prediction=1.000000
(7, apache hadoop) --> prob=[0.982157533344,0.0178424666556], prediction=0.000000
>>>
```

Pipeline:

```
tdend2@node00:~
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.feature import HashingTF, Tokenizer
>>>
```

Prepare training documents from a list of (id, text, label) tuples.

```
tdend2@node00:~
>>> training = spark.createDataFrame([
...     (0, "a b c d e spark", 1.0),
...     (1, "b d", 0.0),
...     (2, "spark f g h", 1.0),
...     (3, "hadoop mapreduce", 0.0)
... ], ["id", "text", "label"])
>>>
```

Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.

```
tdend2@node00:~
>>> tokenizer = Tokenizer(inputCol="text", outputCol="words")

lr = LogisticRegression(maxIter=10, regParam=0.001)
pipel>>> hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
ine = Pipeline(>>> lr = LogisticRegression(maxIter=10, regParam=0.001)
>>> pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
>>>
```

#Fit the pipeline to training documents.

```
tdend2@node00:~
>>> model = pipeline.fit(training)
>>>
```

Prepare test documents, which are unlabeled (id, text) tuples.

```
tdend2@node00:~
>>> test = spark.createDataFrame([
...     (4, "spark i j k"),
...     (5, "l m n"),
...     (6, "spark hadoop spark"),
...     (7, "apache hadoop")
... ], ["id", "text"])
>>>
```

#Make predictions on test documents and print columns of interest.

```
tdend2@node00:~
>>> prediction = model.transform(test)
>>> selected = prediction.select("id", "text", "probability", "prediction")
>>> for row in selected.collect():
...     rid, text, prob, prediction = row
...     print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
...
(4, spark i j k) --> prob=[0.159640773879,0.840359226121], prediction=1.000000
(5, l m n) --> prob=[0.837832568548,0.162167431452], prediction=0.000000
(6, spark hadoop spark) --> prob=[0.0692663313298,0.93073366867], prediction=1.000000
(7, apache hadoop) --> prob=[0.982157533344,0.0178424666556], prediction=0.000000
>>>
```

Above full program code is as shown below:

Estimator, Transformer, and Param

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression

# Prepare training data from a list of (label, features) tuples.
training = spark.createDataFrame([
    (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]))], ["label", "features"])

# Create a LogisticRegression instance. This instance is an Estimator.
lr = LogisticRegression(maxIter=10, regParam=0.01)
# Print out the parameters, documentation, and any default values.
print("LogisticRegression parameters:\n" + lr.explainParams() + "\n")

# Learn a LogisticRegression model. This uses the parameters stored in lr.
model1 = lr.fit(training)

# Since model1 is a Model (i.e., a transformer produced by an Estimator),
# we can view the parameters it used during fit().
# This prints the parameter (name: value) pairs, where names are unique IDs for this
# LogisticRegression instance.
print("Model 1 was fit using parameters: ")
print(model1.extractParamMap())

# We may alternatively specify parameters using a Python dictionary as a paramMap
paramMap = {lr.maxIter: 20}
paramMap[lr.maxIter] = 30  # Specify 1 Param, overwriting the original maxIter.
paramMap.update({lr.regParam: 0.1, lr.threshold: 0.55})  # Specify multiple Params.

# You can combine paramMaps, which are python dictionaries.
paramMap2 = {lr.probabilityCol: "myProbability"}  # Change output column name
paramMapCombined = paramMap.copy()
paramMapCombined.update(paramMap2)

# Now learn a new model using the paramMapCombined parameters.
# paramMapCombined overrides all parameters set earlier via lr.set* methods.
model2 = lr.fit(training, paramMapCombined)
print("Model 2 was fit using parameters: ")
print(model2.extractParamMap())

# Prepare test data
test = spark.createDataFrame([
```

```
    (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]))], ["label", "features"])

# Make predictions on test data using the Transformer.transform() method.
# LogisticRegression.transform will only use the 'features' column.
# Note that model2.transform() outputs a "myProbability" column instead of the usual
# 'probability' column since we renamed the lr.probabilityCol parameter previously.
prediction = model2.transform(test)
result = prediction.select("features", "label", "myProbability", "prediction") \
    .collect()

for row in result:
    print("features=%s, label=%s -> prob=%s, prediction=%s"
          % (row.features, row.label, row.myProbability, row.prediction))
```

Pipeline:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer

# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])

# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])
```

```
# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print("(%d, %s) --> prob=%s, prediction=%f" % (rid, text, str(prob), prediction))
```

—----------

Train a model using the ML pipeline and then test it and show the results. We will use the CIKM2020 AnalytiCup COVID-19 Retweet Prediction Challenge dataset. The CIKM is one of the best conferences in the Machine Learning world.

• The challenge:

• Dataset:

• The challenge goals
o Given the set of features for a tweet from TweetsCOV19, the task is to predict the number of times it will be retweeted (#retweets).
o You may loosen the goal to make it a binary classification problem, e.g., retweeted (#retweets =1+) or not (#retweets =0).

• Loading data (choose one training data from the small sample or the Big full data)
o Training: TweetsCOV19-train.tsv (small sample) or TweetsCOV19.tsv (Big)
o Testing: TweetsCOV19-test-new.tsv (new/unknown)
o HDFS directory: /user/data/CSC534BDA/COVID19-Retweet/

• Features and a target class
o At least one feature and one target class (#retweets)
▪ Input features: At least one, e.g., Entities column (tweets text)
▪ Target class: number of retweets (#retweets column)
▪ You may convert the target class to 1 (#retweets =1+) or 0 (#retweets =0).
o Schema or column names (tap-separated values),
o You may load all features or some features.

• Model: Logistic Regression

```
tdend2@node00:~
[tdend2@node00 ~]$ hdfs dfs -ls /user/data/CSC534BDA/COVID19-Retweet/
Found 4 items
-rw-r--r--   3 sslee777 supergroup      10837 2023-11-12 17:50 /user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-test-new.tsv
-rw-r--r--   3 sslee777 supergroup       6422 2020-10-29 13:29 /user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-test.tsv-rw-r--r--
a/CSC534BDA/COVID19-Retweet/TweetsCOV19-train.tsv
-rw-r--r--   3 sslee777 supergroup 1864393651 2020-10-29 13:26 /user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19.tsv
[tdend2@node00 ~]$
```

Header is not available as seen below:

```
tdend2@node00:~
>>> df = spark.read.option("header", "true").option("delimiter", "\t").csv("/user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-train.tsv")

>>> df.printSchema()
root
 |-- 1178791787386814465: string (nullable = true)
 |-- 35234fe4a19cc1a3336095fb3780bcc1: string (nullable = true)
 |-- Mon Sep 30 22:00:37 +0000 2019: string (nullable = true)
 |-- 619: string (nullable = true)
 |-- 770: string (nullable = true)
 |-- 05: string (nullable = true)
 |-- 06: string (nullable = true)
 |-- null;7: string (nullable = true)
 |-- 2 -1: string (nullable = true)
 |-- null;9: string (nullable = true)
 |-- null;10: string (nullable = true)
 |-- null;11: string (nullable = true)

>>>
```

So, given the schema based on the column names given in the description of the dataset and observing the above first row of the data which is shown below:

Found that there are 12 columns in the dataset including the target class ’retweet’ in which retweet is the 6th column and Entities is the 8th column.

Code:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

schema=StructType([
StructField("tweetID", StringType(), True),
StructField("userHandle", StringType(), True),
```

```
StructField("postedTime", StringType(), True),
StructField("followerCount", StringType(), True),
StructField("friendCount", StringType(), True),
StructField("retweetCount", StringType(), True),
StructField("favouriteCount", StringType(), True),
StructField("tweetEntities", StringType(), True),
StructField("tweetSentiment", StringType(), True),
StructField("userMentions", StringType(), True),
StructField("tweetHashtags", StringType(), True),
StructField("tweetURLs", StringType(), True)])

train_path="/user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-train.tsv"
test_path="/user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-test-new.tsv"

training=spark.read.csv(train_path, schema=schema, sep="\t", header=False)
testing=spark.read.csv(test_path, schema=schema, sep="\t", header=False)

training=training.withColumn("label", (training["retweetCount"]>0).cast("integer"))
testing=testing.withColumn("label", (testing["retweetCount"]>0).cast("integer"))
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="tweetEntities", outputCol="tokens")
hashingTF = HashingTF(inputCol="tokens",outputCol="features")

lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Make predictions on test documents and print columns of interest.
prediction = model.transform(testing)
prediction.select("tweetEntities", "label",  "prediction").show()
```

Execution of above **pipeline code** of covid retweet small dataset for training and test-new dataset for testing and obtained **output**:
Loading libraries and defining schema

```
tdend2@node00:~
>>> from pyspark.ml import Pipeline
ogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.feature import HashingTF, Tokenizer
>>> from pyspark.sql.types import StructType, StructField, IntegerType, StringType
>>> schema=StructType([
... StructField("tweetID", StringType(), True),
... StructField("userHandle", StringType(), True),
... StructField("postedTime", StringType(), True),
... StructField("followerCount", StringType(), True),
... StructField("friendCount", StringType(), True),
... StructField("retweetCount", StringType(), True),
... StructField("favouriteCount", StringType(), True),
... StructField("tweetEntities", StringType(), True),
... StructField("tweetSentiment", StringType(), True),
... StructField("userMentions", StringType(), True),
... StructField("tweetHashtags", StringType(), True),
... StructField("tweetURLs", StringType(), True)])
>>>
```

Loading training training and testing datasets and checking the schema of these:

```
tdend2@node00:~
>>> train_path="/user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-train.tsv"
>>> test_path="/user/data/CSC534BDA/COVID19-Retweet/TweetsCOV19-test-new.tsv"
>>> training=spark.read.csv(train_path, schema=schema, sep="\t", header=False)
>>> training.printSchema()
root
 |-- tweetID: string (nullable = true)
 |-- userHandle: string (nullable = true)
 |-- postedTime: string (nullable = true)
 |-- followerCount: string (nullable = true)
 |-- friendCount: string (nullable = true)
 |-- retweetCount: string (nullable = true)
 |-- favouriteCount: string (nullable = true)
 |-- tweetEntities: string (nullable = true)
 |-- tweetSentiment: string (nullable = true)
 |-- userMentions: string (nullable = true)
 |-- tweetHashtags: string (nullable = true)
 |-- tweetURLs: string (nullable = true)
```

```
tdend2@node00:~
>>> testing=spark.read.csv(test_path, schema=schema, sep="\t", header=False)
>>> testing.printSchema()
root
 |-- tweetID: string (nullable = true)
 |-- userHandle: string (nullable = true)
 |-- postedTime: string (nullable = true)
 |-- followerCount: string (nullable = true)
 |-- friendCount: string (nullable = true)
 |-- retweetCount: string (nullable = true)
 |-- favouriteCount: string (nullable = true)
 |-- tweetEntities: string (nullable = true)
 |-- tweetSentiment: string (nullable = true)
 |-- userMentions: string (nullable = true)
 |-- tweetHashtags: string (nullable = true)
 |-- tweetURLs: string (nullable = true)

>>> training=training.withColumn("label", (training["retweetCount"]>0).cast("integer"))
>>> testing=testing.withColumn("label", (testing["retweetCount"]>0).cast("integer"))
>>> tokenizer = Tokenizer(inputCol="tweetEntities", outputCol="tokens")
```

Created a new column ‘label’ to show if retweeted or not as shown above and configured an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr as shown below.
Fit the pipeline to training documents. Made predictions on testing dataset.

```
tdend2@node00:~
>>> hashingTF = HashingTF(inputCol="tokens",outputCol="features")
>>> lr = LogisticRegression(maxIter=10, regParam=0.001)
>>> pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
>>> model = pipeline.fit(training)
24/11/04 13:55:50 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
24/11/04 13:55:50 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
>>> prediction = model.transform(testing)
>>> prediction.select("tweetEntities", "label",  "prediction").show()
+--------------------+-----+----------+
|       tweetEntities|label|prediction|
+--------------------+-----+----------+
|neoliberals:Neoli...|    0|       0.0|
|               null;|    1|       0.0|
|nazi:Nazism:-2.74...|    1|       0.0|
|               null;|    1|       0.0|
|immune system:Imm...|    0|       1.0|
|the joker:Joker_%...|    0|       1.0|
|puberty:Puberty:-...|    0|       0.0|
|coconut milk:Coco...|    0|       1.0|
|afp:Agence_France...|    1|       0.0|
|abc:American_Broa...|    0|       0.0|
|               null;|    0|       0.0|
|               null;|    0|       0.0|
|lip:Lip_%28gastro...|    0|       1.0|
|e 40:E-40:-0.8783...|    1|       1.0|
|the topper:The_To...|    1|       1.0|
|riot police:Riot_...|    0|       0.0|
|ballum:Ballum:-2....|    1|       0.0|
|               null;|    0|       0.0|
|hong kong:Hong_Ko...|    1|       1.0|
|bon appetit:Bon_A...|    0|       1.0|
+--------------------+-----+----------+
only showing top 20 rows

>>>
```

From the above output, labels are seen as 0 and 1. 1 indicates that tweet entities are retweeted and 0 as viceversa. Also found misclassifications in the predictions.

==========THE END==========