COMP5349 –Cloud Computing

Week 9: Spark Machine Learning

Dr. Ying Zhou
School of Computer Science

Last Week

- Data sharing in Spark
  - Closure
  - Broadcast Variables
  - Accumulators
  - RDD persistence
- Spark SQL and the DataFrame API
  - DataFrame is easier to manipulate
    - Access to individual columns
  - More efficient with building optimization mechanism
  - DataFrame is also immutable
  - DataFrame and RDD can be converted to each other

Outline

- Spark Machine Learning
  - Machine Learning Libraries
  - Basic Data Types
  - Standardized API

- Assignment Brief Intro

Spark Machine Learning Libraries

- Spark has two versions of Machine Learning library
  - RDD based
    - Package/module name **mllib**
  - DataFrame based
    - Package/name name **ml**
    - The primary API with potential new features introduced in new release
- Most machine learning algorithms expect data in **vector** or **matrix** form
  - Both libraries provide data types representing those, make sure you always use the one in **ml** package

Basic Data Types

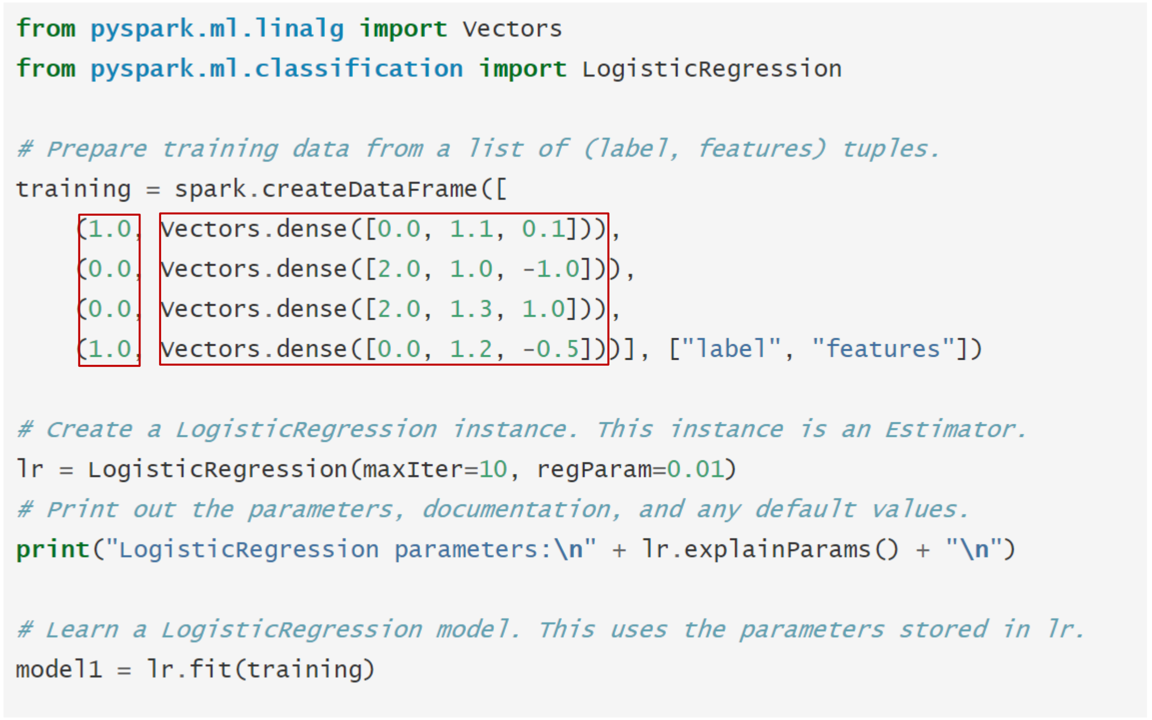
- The linear algebra sub package/module **linalg** defines basic data structure used in most machine learning algorithms
- **Vector** is the general class and it is a *local* data type
  - We can have RDDs with Vector as part of its element
    - **RDD**: (Integer, (String, Vector))
  - We can have DataFrame with Vector as its column
    - **DataFrame**: (Integer, String, Vector, Vector)

Vectors

- Concrete vectors can be expressed in two forms
  - ***Dense vector*** is like an array, it may be implemented using different structures in different languages
  - ***Sparse vector*** is an efficient way to save vectors with many zeros
    - It remembers the size of the vector and all non-zero (index, value) pair.
  - e.g. the vector (0.5, 0.0, 0.3)
    - Dense format: [0.5, 0.0, 0.3]
    - Sparse format: (3,{0:0.5},{2:0.3}) or (3,[0,2],[0.5,0.3])
- Spark provides its own type hierarchy for **`Vector`** to have a consistent API across and also to provide some useful methods on it.
  - dot product of two vectors
  - squared distance of two vectors

DataFrame of Vectors

- Many algorithms expect input data to contain a particular column of type **`Vector`**, this column is usually called ‘features’

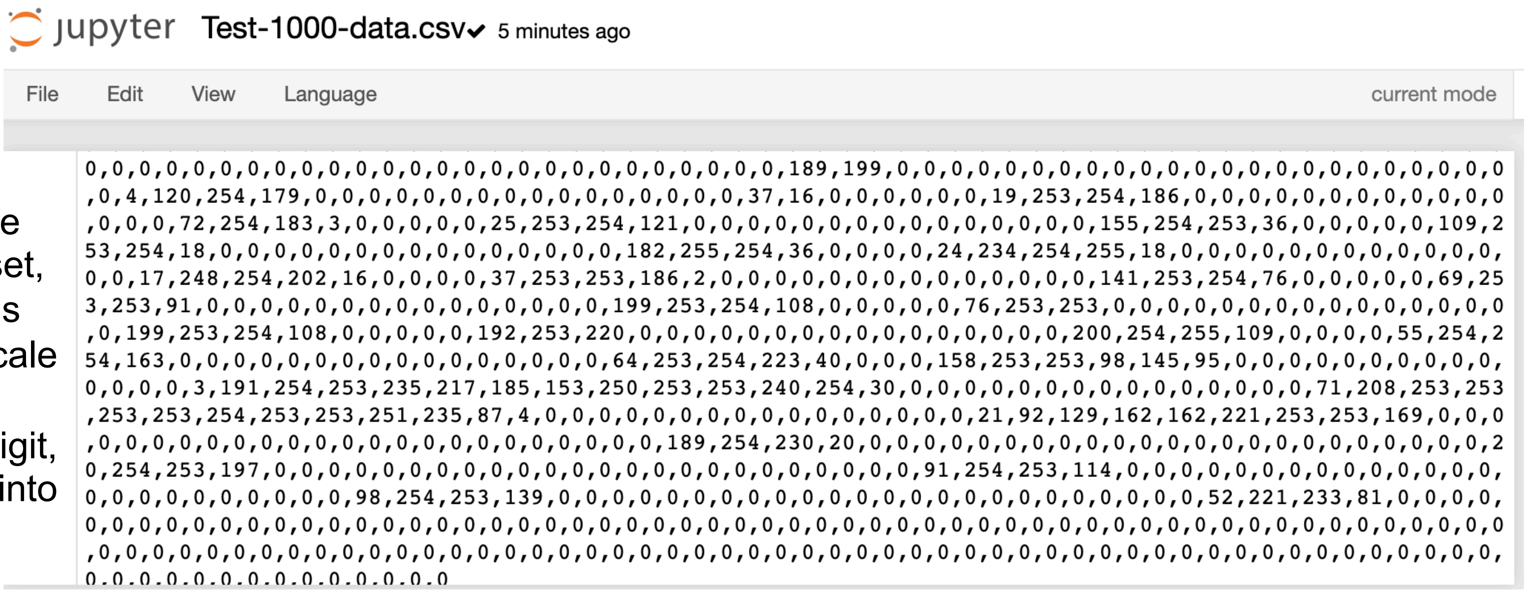


```python
from pyspark.ml.linalg import Vectors
from pyspark.ml.classification import LogisticRegression

# 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)
```

Creating Vectors from stored data

- Most data set are stored as CSV or text file, where the values are separated by some delimiters
- The default read mechanism would read those into a data frame with many columns, each representing a single value
- Spark provides a utility called **VectorAssembler** that can merge specified columns to create a new column of Vector type.

One row of the MNIST data set, which contains 28x28 gray scale image for handwritten digit, It is flattened into a 784 vector



```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,189,199,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,4,120,254,179,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,37,16,0,0,0,0,0,0,19,253,254,186,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,72,254,183,3,0,0,0,0,25,253,254,121,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,155,254,253,36,0,0,0,0,0,109,2
53,254,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,182,255,254,36,0,0,0,0,24,234,254,255,18,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,17,248,254,202,16,0,0,0,0,37,253,253,186,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,141,253,254,76,0,0,0,0,0,69,25
3,253,91,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,199,253,254,108,0,0,0,0,0,76,253,253,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,199,253,254,108,0,0,0,0,0,192,253,220,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,200,254,255,109,0,0,0,0,55,254,2
54,163,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,64,253,254,223,40,0,0,0,158,253,253,98,145,95,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,3,191,254,253,235,217,185,153,250,253,253,240,254,30,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,71,208,253,253
,253,253,254,253,253,251,235,87,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,21,92,129,162,162,221,253,253,169,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,189,254,230,20,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2
0,254,253,197,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,91,254,253,114,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,98,254,253,139,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,52,221,233,81,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

VectorAssembler usage

```
test_datafile = "file:///home/Test-1000-data.csv"
test_labelfile= "file:///home/Test-1000-label.csv"

num_test_samples = 1000

test_df = spark.read.csv(test_datafile,header=False,inferSchema="true")
```

```
assembler = VectorAssembler(inputCols=test_df.columns,
    outputCol="features")
test_vectors = assembler.transform(test_df).select("features")
test_vectors.show(2)
```

```
+--------------------+
|            features|
+--------------------+
|(784,[202,203,204...|
|(784,[94,95,96,97...|
+--------------------+
only showing top 2 rows
```

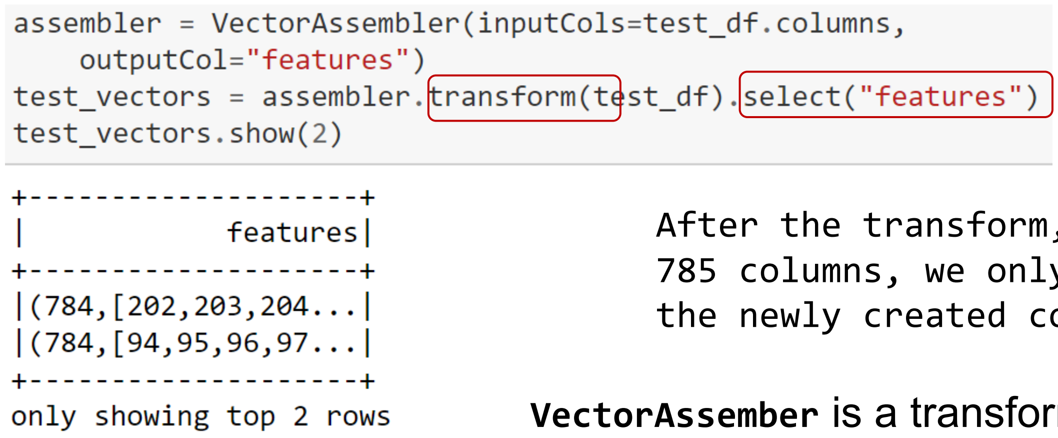
Sparse vector format

Machine Learning Algorithms

- Many common algorithms are implemented in SparkML and are grouped into a few package/modules
  - Clustering
  - Classification
  - Feature
  - Etc..
- The DataFrame based library use standardized API for all algorithms
- Many analytic problems may need more than one algorithms, they can be combined into a single pipeline
- Most of the concepts are inspired or borrowed from **scikit-learn**

Pipeline Component: Transformers

- **`Transformer`**
  - An abstract term that include feature transformer and learned models.
  - A **`transformer`** implements a method **`transform()`**, which converts one DataFrame (input) into another(output), generally by appending one or more columns (to indicate a prediction or other results)



```
assembler = VectorAssembler(inputCols=test_df.columns,
    outputCol="features")
test_vectors = assembler.transform(test_df).select("features")
test_vectors.show(2)
```

```
+--------------------+
|            features|
+--------------------+
|(784,[202,203,204...|
|(784,[94,95,96,97...|
+--------------------+
only showing top 2 rows
```

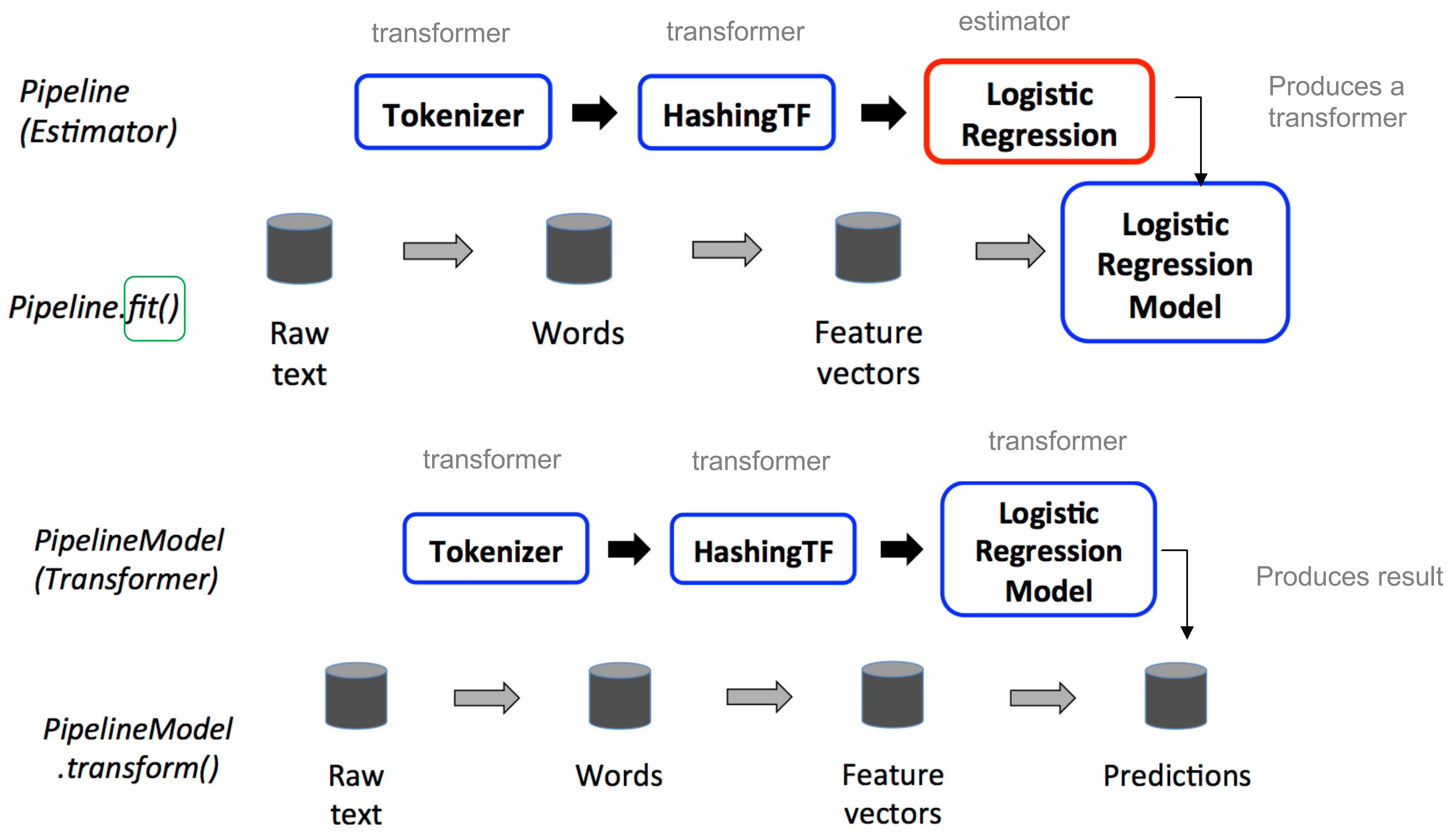
`After the transform, we get a DF with 785 columns, we only want to retain the newly created column`

`VectorAssember` is a transformer, doing relatively simple task by assembling specified input columns into a single output column of Vector type

Pipeline Component: Estimators and Pipeline

- Estimators
  - **`Estimator`** abstracts the concept of a learning algorithm or any algorithm that fits or trains on data.
  - An **`Estimator`** implements a method **fit()**, which accepts a **`DataFrame`** (training set) and produces a **`Model`**, which is a **`Transformer`**
  - The actual model's presentation would be very different in different algorithms
- Pipeline
  - Pipeline represents a workflow consisting of a sequence of stages (**`transformers`** and **`estimators`**)
  - Pipeline usually takes a **`DataFrame`** as an input and produce a model as output
  - Once built and trained, a pipeline can be used to transform data

Example Pipeline



Pipeline Code Example

```python
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"])
```

Id	Text	label
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

Pipeline Code Example

```
# 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])
```

Id	Text	label	words	features
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		

DataFrame is also immutable

Pipeline Code Example

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

```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
```

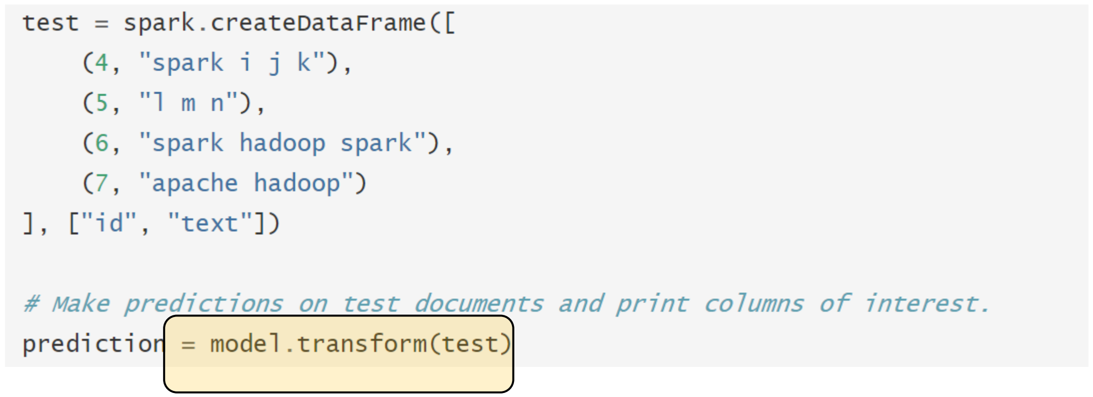
```
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
```

Id	Text	label	words	features
0	“a b c d e spark”	1.0	[“a”, “b”,..]	[1,1,…]
1	“b d”	0.0	[“b”, “d”,…]	[0,1,…]
2	“spark f g h”	1.0	[“spark”, “f”, …]	[0,0,…]
3	“Hadoop mapreduce”	0.0	[“Hadoop”, “mapreduce]	[0,0,…]

It output a model, which includes all transformers in the pipeline as well as logistic function

A few intermediate DataFrames are generated,

Pipeline Code Example

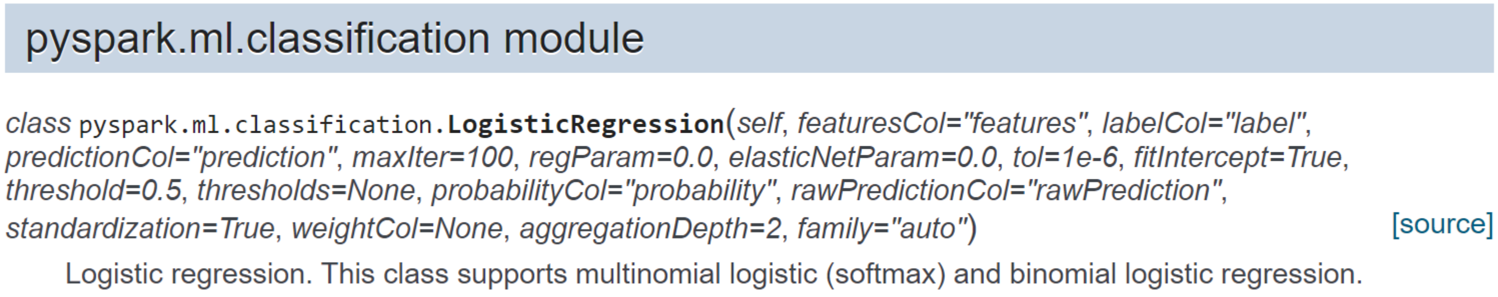


```
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])

# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
```

Id	Text	words	features	probability	prediction
4	“spark i j k”	[spark, i, ..]	[0,0,..]	[0.16,0.84]	1
5	“l m n”	[l,m,n]	[0,0,…]	[0.84,0.16]	0
6	“spark Hadoop spark”	[spark,…]	[0,0,…]	[0.07,0.93]	1
7	“apache hadoop”	[apache,…]	[0,0,…]	[0.98,0.02]	0

Parameters

- Different algorithms take different parameters

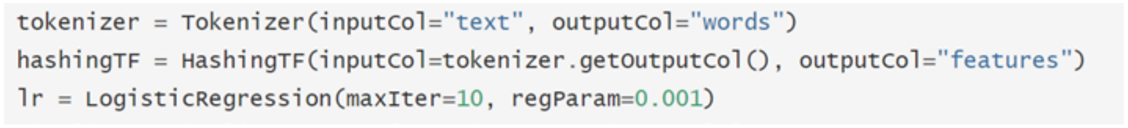


pyspark.ml.classification module

class `pyspark.ml.classification.`**`LogisticRegression`**(*self, featuresCol="features", labelCol="label", predictionCol="prediction", maxIter=100, regParam=0.0, elasticNetParam=0.0, tol=1e-6, fitIntercept=True, threshold=0.5, thresholds=None, probabilityCol="probability", rawPredictionCol="rawPrediction", standardization=True, weightCol=None, aggregationDepth=2, family="auto"*) [source]

Logistic regression. This class supports multinomial logistic (softmax) and binomial logistic regression.

- Parameters can be specified as literals when creating algorithm instances



```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
```

- They can be put together in a hashmap like structure **`ParaMap`** and passed to algorithm instances

```
ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)
```

Outline

- Spark Machine Learning
  - Machine Learning Libraries
  - Basic Data Types
  - Standardized API

- **Assignment Brief Intro**
  - **Various ways of using Spark in machine learning workload**
  - **Assignment intro**
  - **Next week's lab**

Various options of using Spark in ML

- Both MapReduce and Spark are good candidate for Exploratory Data Analytic workload
- Spark is also designed for predictive workload, either with basic statistic model or with machine learning model
- There are various options of using Spark in ML
  - Use Spark RDD and SQL to explore or prepare the input data
  - Use Spark ML API to perform predictive analysis
  - Using Spark and external ML packages
  - Writing customized Spark ML algorithm
- The assignment covers first three options
- Week 10 lab demonstrates the second and third option
  - Based on week 9 lecture

Assignment Brief Intro

- Data Set:
  - Text corpus: Multi-Genre Natural Language Inference (**MultiNLI**).
  - Each data point is a pair of sentences: a *premise* and a *hypothesis*
  - Sentences are selected from 10 domains (genres)
- Tasks
  - Explore vocabulary coverage among genres
    - Mostly Spark RDD and/or Spark SQL
  - Explore and compare sentence vector representation methods
    - SparkML together with Spark RDD and/or Spark SQL

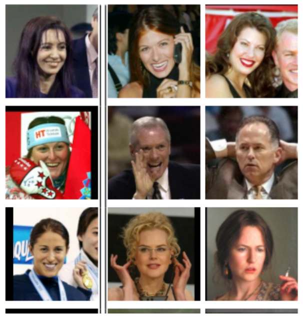
Representation Generation

- Embedding or representation learning is a technique to convert some form of input into a *fixed sized vector*
  - The output vectors can be used as feature vectors in traditional machine learning algorithms or in deep learning algorithms
  - The input can be of various format
  - Many ML algorithms requires measuring distance/similarity between data points, a vector representation provides many options for distance computation
  - Deep Learning model has been proved effective in representation generation

224 x 224 x 3

Any image classification model

4096



220 × 220 × 3

FaceNet by Google (2015)

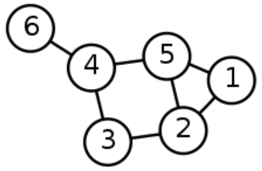
128

… ,School of computer science is formerly called school of information technologies ,....

TFIDF or similar

Word2Vec, Sentence Encoder

100



Node2Vec or similar

128

Spark Provided Representation Generation

- SparkML provides some representation generation mechanisms for text data under the category “Feature Extractors”
  - **TF-IDF**
  - Word2Vec
  - CountVectorizer
  - FeatureHasher

Spark Feature Extractor: TF-IDF

- Term frequency – Inverted Document Frequency
  - Each document is represented by a fixed ***d***-dimensional vector
  - Each dimension represents a term(word) in the vocabulary
  - The value is computed using the frequency of that word in the document and the inversed document frequency of that word in the corpus.

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1},$$

- Term frequency and word-index mapping
  - CountVectorizer
    - Maintain a term-index mapping: {term1:0, term2:1}
    - Value *d* could be the vocabulary size or a smaller given number
    - When the value *d* is less than vocab size, the top *d* frequent terms are used.
  - HashingTF
    - Use a hash function to map term to index
    - No need to maintain a large dictionary
    - Value *d* should be larger than the vocab size to avoid hash collision

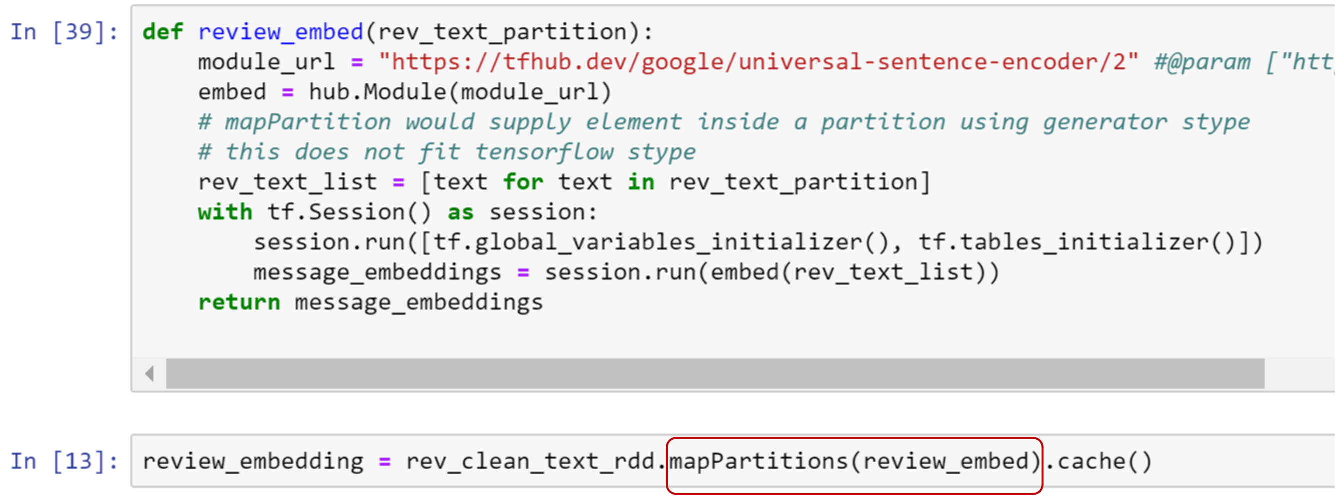
Spark and Third Party Representation Generation

- Spark can be used together with a trained model to generate features
  - Embarrassingly parallel workload
  - We can run the generation tasks in parallel on different partition of the data set on multiple nodes
  - The generation model can be treated as a black box, we only need to know how to prepare input and collect output
- Spark has many **`map`** like operators (e.g. **`map`**, **`filter`**, **`flatMap`**, **`mapToPair)`**, it also has a **`mapPartition`** operator
  - **`Map`**, **`filter`**, **`flatMap`**, etc operate on a single element in the partition and produce an output for each element
  - **`mapPartition`** operates on the whole partition and generate one output for the whole partition
- In representation generation, we need to apply the model on each input element, but the model itself is usually large, it is more efficient to load it once and apply to many inputs

Week 10 Lab

- The lab exercises use two typical data sources in machine learning
  - Image and text
- Basic usage of Spark Machine Learning API is demonstrated with the MNIST image data set
- Example showing how to use Spark with deep learning models to generate sentence encoding vector.

Spark and Representation Generation



```
In [39]: def review_embed(rev_text_partition):
             module_url = "https://tfhub.dev/google/universal-sentence-encoder/2" #@param ["htt
             embed = hub.Module(module_url)
             # mapPartition would supply element inside a partition using generator stype
             # this does not fit tensorflow stype
             rev_text_list = [text for text in rev_text_partition]
             with tf.Session() as session:
                 session.run([tf.global_variables_initializer(), tf.tables_initializer()])
                 message_embeddings = session.run(embed(rev_text_list))
             return message_embeddings
```

```
In [13]: review_embedding = rev_clean_text_rdd.mapPartitions(review_embed).cache()
```

References

- Spark Machine learning library documentation
  - https://spark.apache.org/docs/latest/ml-pipeline.html