

B I G D A T A

C U R S 11

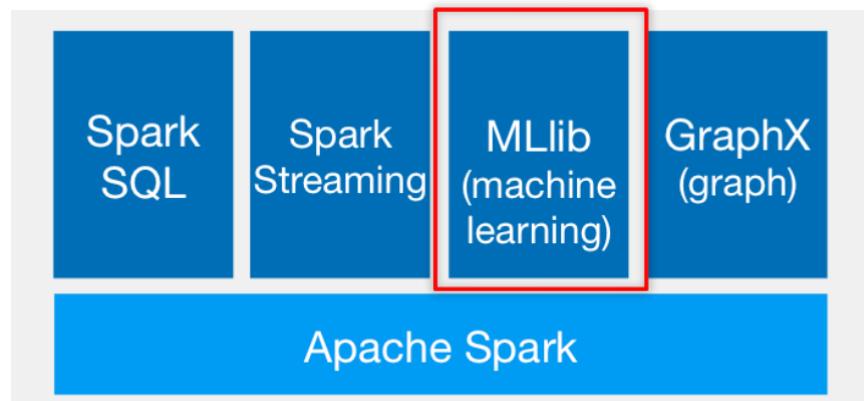




Apache Spark MLlib

Introducere

Apache Spark MLlib



Apache Spark MLlib

- MLlib – biblioteca de *machine learning* (scalabilă) a lui Apache Spark
- **Simplu de folosit** în Java, Scala, Python, R
 - Se încadrează în API-urile Spark și interoperează cu NumPy
 - Se pot folosi sursele de date Hadoop (HDFS, Hbase, fișiere locale) => este posibilă integrarea în fluxurile de lucru Hadoop

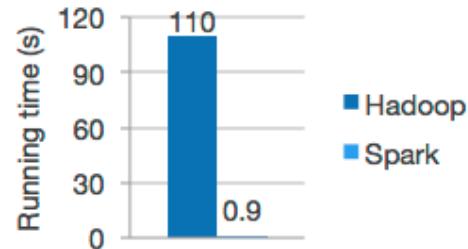
```
data = spark.read.format("libsvm")\
    .load("hdfs://...")

model = KMeans(k=10).fit(data)
```

Apel MLlib în Python

Apache Spark MLlib

- **Performanța** MLlib este determinată de performanța motorului Spark
 - Spark excelează în privința calculului interativ => MLlib rulează rapid
 - Pe de altă parte, MLlib asigură performanța algoritmilor implementați:
 - Algoritmi eficienți care utilizează iterația, ce pot avea rezultate mai bune decât aproximările de tip one-pass ce sunt folosite uneori în MapReduce.



Regresie logistică în Spark vs Hadoop

Apache Spark MLlib

- Rulează pe diferite medii și pe diferite surse de date
 - Medii: Hadoop, Apache Mesos, Kubernetes, *standalone, cloud*
 - Sursele de date pe care le poate accesa includ: HDFS, Apache Cassandra, Apache Hbase, Apache Hive etc...
- **MLlib** conține numeroase funcționalități

Apache Spark MLlib

Funcționalitățile MLlib:

- Algoritmi ML: algoritmii comuni de învățare (clasificare, regresie, clustering, filtrare colaborativă)
- Prelucrarea caracteristicilor: extragere, transformare, reducerea dimensiunii, selecție
- *Pipelines*: tool-uri pentru construirea, evaluarea și optimizarea *pipeline*-urilor ML
- Persistență: salvarea și încărcarea algoritmilor, modelelor și a *pipeline*-urilor
- Utilitară: algebra liniară, statistică, prelucrarea datelor etc.

Algoritmi MLlib

- Clasificare: regresie logistică, naive Bayes etc.
- Regresie: regresie liniară generalizată, regresie de supraviețuire etc.
- Arboi de decizie, *random forests*, arbori *gradient-boosted*
- Recomandare: ALS (*Alternating Least Squares*)
- Clustering: K-means, GMM (*Gaussian Mixtures*) etc.
- Modelarea topic-urilor: LDA (*Latent Dirichlet Allocation*)
- Reguli de asociere, seturi de *item*-uri frecvente, *mining* de *pattern*-uri

API Apache Spark MLlib

- **API bazat pe DataFrame** (pachetul `spark.ml`) – API-ul principal de ML pentru Spark
- **API bazat de RDD** (pachetul `spark.mllib`) – se află în mod menținanță (versiunea ≥ 2.0)
 - MLlib suportă încă continuare acest API
 - Corectează bug-uri, dar nu adaugă funcționalități noi
- De ce API bazat pe DataFrame?
 - API-ul DataFrame – mai *user-friendly* decât RDD
 - Beneficii DataFrame: surse de date, interogări SQL, optimizare (Catalyst, Tungsten), uniformitatea API-ului în diferitele limbaje în care este disponibil
 - Structurile de tip DataFrame facilitează *pipeline*-urile ML, în particular transformarea caracteristicilor

Spark ML

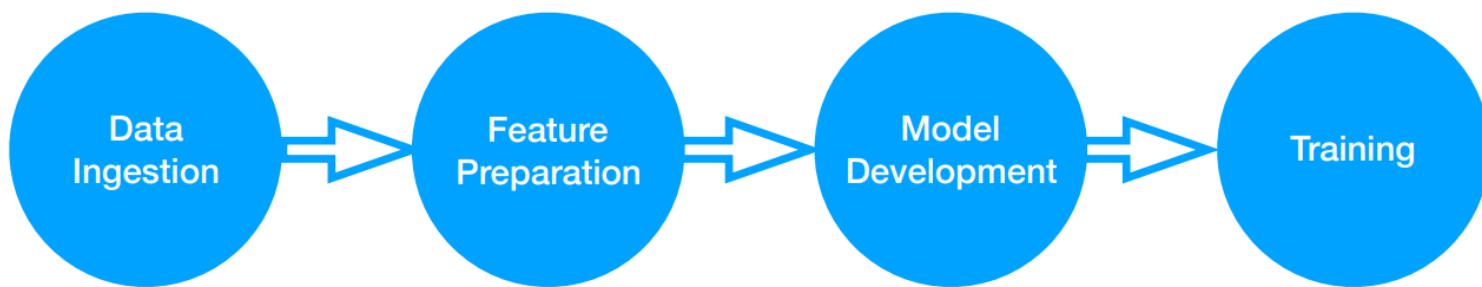
- Termen neoficial, ce se referă la API-ul MLlib bazat pe DataFrame – inspirat de:
 - Numele pachetului spark.ml
 - Conceptul Spark ML Pipeline

Dependențe

- Pentru procesarea numerică optimizată, MLlib folosește pachetele de algebră liniară:
 - Breeze
 - Netlib-java
- Aceste pachete pot apela librării native de accelerare, dacă acestea sunt disponibile. Din motive de licențiere, *proxy*-urile native netlib-java nu pot fi distribuite împreună cu Spark.
- Procesarea accelerată a algebrei linare este o funcționalitate ce poate fi activată.
- Dacă nu este activată vor apărea câteva mesaje de tip *warning*

ML Pipeline

Introducere



- Un exemplu:
<https://indico.cern.ch/event/761215/contributions/3158971/attachments/1724310/2784714/SparkPipeline.pdf>

Introducere

- Un proiect ML are mai multe componente ce trebuie legate între ele
- Etape:
 - Efectuarea de operații de bază pe obiecte DataFrame (citire, definire schemă)
 - Explorarea datelor (determinarea dimensiunii datelor, descrierea datelor, numărul de valori necunoscute, numărul de valori unice dintr-o coloană etc.)
 - Encodarea variabilelor categoriale (*String Indexing, One Hot Encoding*)
 - Asamblarea vectorilor
 - Crearea *pipeline*-ului
- Un *pipeline* conține
 - Transformatori și estimatori
- Exemplu: Vom ilustra etapele de mai sus pe *dataset*-ul
<https://drive.google.com/file/d/1wUcLfGZ2HxvJNstZwM1e2CeNDTrJLy9b/view>

Etapa 1 – Operații de bază pe DataFrame

- Un prim pas esențial într-un proiect de data science este **înțelegerea datelor** – înainte de a construi modelele asupra lor.
- Trebuie înțelese variabilele (atributele) setului de date. În setul de date exemplu:
 - **Batsman:** Codul unic al jucătorilor de tip *batsman* (Integer)
 - **Batsman_Name:** Numele jucătorilor de tip *batsman* (String)
 - **Bowler:** Codul unic al jucătorilor de tip *bowler* (Integer)
 - **Bowler_Name:** Numele jucătorilor de tip *bowler* (String)
 - **Commentary:** Descrierea evenimentului, aşa cum este furnizată în cadrul transmisiunii (String)
 - **Detail:** Descrierea evenimentelor precum cele de tip *wickets* și *extra deliveries* (String)
 - **Dismissed:** Codul jucătorului *batsman* eliminat (String)
 - **Id:** Codul unic al înregistrării (liniei) (String)
 - **Isball:** Indică dacă un *delivery* a fost legal (Boolean)
 - **Isboundary:** Indică dacă un *batsman* a atins marginea (Binary)
 - **Iswicket:** Indică dacă un *batsman* a lovit *wicket-ul* (Binary)
 - **Over:** Numărul de *over-uri* (Double)
 - **Runs:** *Run-uri* în cadrul unui *delivery* (Integer)
 - **Timestamp:** Data și ora la care a fost înregistrat evenimentul (Timestamp)

Etapa 1 – Operații de bază pe DataFrame

- **Citirea** fișierului CSV
- Observarea tipurilor de date (cu ajutorul funcției *printSchema*) – implicit, toate sunt String

```
# Citirea fișierului
my_data = spark.read.csv('ind-ban-comment.csv', header=True)

# Observarea schemei implicate
my_data.printSchema()
```

Etapa 1 – Operații de bază pe DataFrame

- **Definirea schemei:**
 - Nu dorim ca toate coloanele din *dataset* să fie tratate ca *string*-uri
 - Definim schema corespunzătoare: creăm un obiect *StructType* pe baza unei liste de obiecte de tip *StructField*
 - Definiția fiecărui *StructField* va conține un nume de coloană (*name*), tipul de date al coloanei (*dataType*) și dacă aceasta permite valori *null* (*nullable*)

Etapa 1 – Operații de bază pe DataFrame

```
import pyspark.sql.types as tp

# Definirea schemei
my_schema = tp.StructType([
    tp.StructField(name= 'Batsman',           dataType= tp.IntegerType(), nullable= True),
    tp.StructField(name= 'Batsman_Name',       dataType= tp.StringType(), nullable= True),
    tp.StructField(name= 'Bowler',             dataType= tp.IntegerType(), nullable= True),
    tp.StructField(name= 'Bowler_Name',        dataType= tp.StringType(), nullable= True),
    tp.StructField(name= 'Commentary',          dataType= tp.StringType(), nullable= True),
    tp.StructField(name= 'Detail',              dataType= tp.StringType(), nullable= True),
    tp.StructField(name= 'Dismissed',           dataType= tp.IntegerType(), nullable= True),
    tp.StructField(name= 'Id',                  dataType= tp.IntegerType(), nullable= True),
    tp.StructField(name= 'Isball',              dataType= tp.BooleanType(), nullable= True),
    tp.StructField(name= 'Isboundary',          dataType= tp.BinaryType(), nullable= True),
    tp.StructField(name= 'Iswicket',             dataType= tp.BinaryType(), nullable= True),
    tp.StructField(name= 'Over',                dataType= tp.DoubleType(), nullable= True),
    tp.StructField(name= 'Runs',                dataType= tp.IntegerType(), nullable= True),
    tp.StructField(name= 'Timestamp',            dataType= tp.TimestampType(), nullable= True)
])

# Citirea datelor conform noii schemei
my_data = spark.read.csv('ind-ban-comment.csv', schema= my_schema, header= True)

# Afisarea schemei
my_data.printSchema()
```

Etapa 1 – Operații de bază pe DataFrame

- **Ștergerea unor coloane**
 - În seturile de date există adeseori coloane care nu sunt necesare în rezolvarea problemei
 - Ștergem coloanele nefolositoare cu ajutorul funcției *drop*
 - Caracterul „*” care precede lista semnifică ștergerea de coloane multiple din setul de date

```
# Ștergerea coloanelor care nu intervin în proiect
my_data = my_data.drop(['Batsman', 'Bowler', 'Id'])
# Afisarea coloanelor care au ramas
my_data.columns
```

Etapa 2 – Explorarea datelor

- **Determinarea dimensiunii datelor**

- În Pandas DataDrame există funcția *shape* ce permite obținerea dimensiunii datelor
- În Spark DataFrame:

```
# Dimensiunea datelor:  
(my_data.count() , len(my_data.columns))  
# 605 lini, 11 coloane
```

Etapa 2 – Explorarea datelor

- **Descrierea datelor**

- Funcția *describe* returnează valorile statistice de bază (medie, *count*, min, max, deviația standard)
- Funcția *summary* returnează și cuartilele variabilelor numerice:

```
# Sumarul coloanelor numerice:  
my_data.select('Isball', 'Isboundary', 'Runs').describe().show()
```

Etapa 2 – Explorarea datelor

- **Numărul valorilor necunoscute**
 - Frecvent, din seturile de date lipsesc valori
 - Este important să cunoaștem numărul de valori necunoscute din fiecare coloană, pentru a le trata înainte de a construi modele ce utilizează datele respective.

```
# Importul funcțiilor Spark SQL
import pyspark.sql.functions as f

# Valorile null din fiecare coloană
data_agg = my_data.agg(*[f.count(f.when(f.isnull(c),
c)).alias(c) for c in my_data.columns])
data_agg.show()
```

Etapa 2 – Explorarea datelor

- **Obținerea numărului de valori din coloane**
 - În Pandas DataFrame există funcția *value_counts()*
 - Putem folosi funcția *groupBy* pentru a obține numărul de valori unice ale variabilelor categoriale:

```
# Numărul de valori din coloana Batsman_Name  
my_data.groupBy('Batsman_Name').count().show()
```

Etapa 3 – Encodarea variabilelor categoriale

- Majoritatea algoritmilor ML acceptă doar date numerice => este important să convertim variabilele categoriale prezente în setul de date în numere
- Evident, nu putem șterge aceste valori din setul de date fiindcă ele pot conține informații valoroase
- Metode de encodare în Spark: *String Indexing, One Hot Encoding*

Etapa 3 – Encodarea variabilelor categoriale

- ***String Indexing*** – atribuie un număr întreg unic fiecărei categorii
 - 0 este atribuit categoriei celei mai frecvente, 1 următoarei celei mai frecvente categorii etc.
- Trebuie să specificăm coloana de intrare (ce va fi indexată) și cea de ieșire (în care este furnizat rezultatul)

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder

# Crearea unui obiect StringIndexer, specificarea coloanelor de intrare și ieșire
SI_batsman = StringIndexer(inputCol='Batsman_Name',outputCol='Batsman_Index')
SI_bowler = StringIndexer(inputCol='Bowler_Name',outputCol='Bowler_Index')

# Transformarea datelor
my_data = SI_batsman.fit(my_data).transform(my_data)
my_data = SI_bowler.fit(my_data).transform(my_data)

# Afisarea datelor transformate
my_data.select('Batsman_Name', 'Batsman_Index', 'Bowler_Name', 'Bowler_Index').show(10)
```

Etapa 3 – Encodarea variabilelor categoriale

- ***One-Hot Encoding***
- *OneHotEncoder* din Spark nu encodează variabilele categoriale direct.
 - Trebuie folosit mai întâi *StringIndexer* pentru convertirea variabilei în tip numeric
 - Apoi se utilizează *OneHotEncoder* pentru a encoda coloane multiple din setul de date
 - Se creează un vector rar pentru fiecare linie

```
# crearea obiectului si specificarea coloanelor de intrare si iesire
OHE = OneHotEncoder(inputCols=['Batsman_Index',
'Bowler_Index'],outputCols=['Batsman_OHE', 'Bowler_OHE'])

# transformarea datelor
my_data = OHE.fit(my_data).transform(my_data)

# vizualizarea si transformarea datelor
my_data.select('Batsman_Name', 'Batsman_Index', 'Batsman_OHE',
'Bowler_Name', 'Bowler_Index', 'Bowler_OHE').show(10)
```

Etapa 4 – Asamblarea vectorilor

- Un *vector assembler* combină o listă dată de coloane într-un singur vector coloană.
- Această operație are loc de obicei la finalul etapelor de explorare a datelor și preprocesare a acestora
- În acest moment, se lucrează de obicei cu un număr mai mic de caracteristici, aflate fie în forma lor brută fie în cea transformată, ce pot fi folosite pentru antrenarea modelului.
- Asamblorul de vectori le convertește într-o singură coloană de caracteristici pentru a antrena modelul ML (de exemplu, cu ajutorul regresiei logistice).
- Acceptă coloane de tip numeric, boolean sau vector

Etapa 4 – Asamblarea vectorilor

```
from pyspark.ml.feature import VectorAssembler

# Se specifica coloanele de intrare si iesire ale asamblorului de
vectori
assembler = VectorAssembler(inputCols=['Isboundary',
                                         'Iswicket',
                                         'Over',
                                         'Runs',
                                         'Batsman_Index',
                                         'Bowler_Index',
                                         'Batsman_OHE',
                                         'Bowler_OHE'],
                             outputCol='vector')

# se completeaza valorile null
my_data = my_data.fillna(0)

# se transforma datele
final_data = assembler.transform(my_data)

# afisarea vectorului transformat
final_data.select('vector').show()
```

Etapa 5 – Construirea *ML Pipeline*

- De obicei, un proiect ML include pași precum preprocesarea datelor, extragerea caracteristicilor, antrenarea modelului (*model fitting*) și evaluarea rezultatelor.
- => Vor fi efectuate o mulțime de transformări asupra datelor, într-o anumită ordine
- Aceste transformări pot fi greu de gestionat
- Un *Pipeline* permite menținerea fluxului de date prin toate transformările relevante, necesare pentru obținerea rezultatului final.
 - Trebuie definite etapele *pipeline*-ului, ce vor constitui de fapt un lanț de comenzi ce va trebui executate (de către Spark)
 - Fiecare etapă este fie un transformator, fie un estimator.

Etapa 5 – Construirea *ML Pipeline*

- **Transformatori și estimatori**
 - Transformatorii convertesc un DataFrame în altul fie prin actualizarea valorilor curente ale unei coloane (de exemplu, prin convertirea coloanelor categoriale în coloane numerice) fie prin maparea acestora în alte valori (conform unor cerințe definite).
 - Un estimator implementează metoda *fit* asupra unui DataFrame și produce un model.
 - De exemplu, *LogisticRegression* este un estimator ce antrenează un model de clasificare atunci când apelăm metoda *fit*.

Etapa 5 – Construirea *ML Pipeline*

- **Exemplul 1**

- Considerăm un dataframe cu 3 coloane având valorile de mai jos.
- Definim unele dintre etapele în cadrul cărora dorim să transformăm datele și, cu ajutorul acestora, vom crea un *pipeline*:

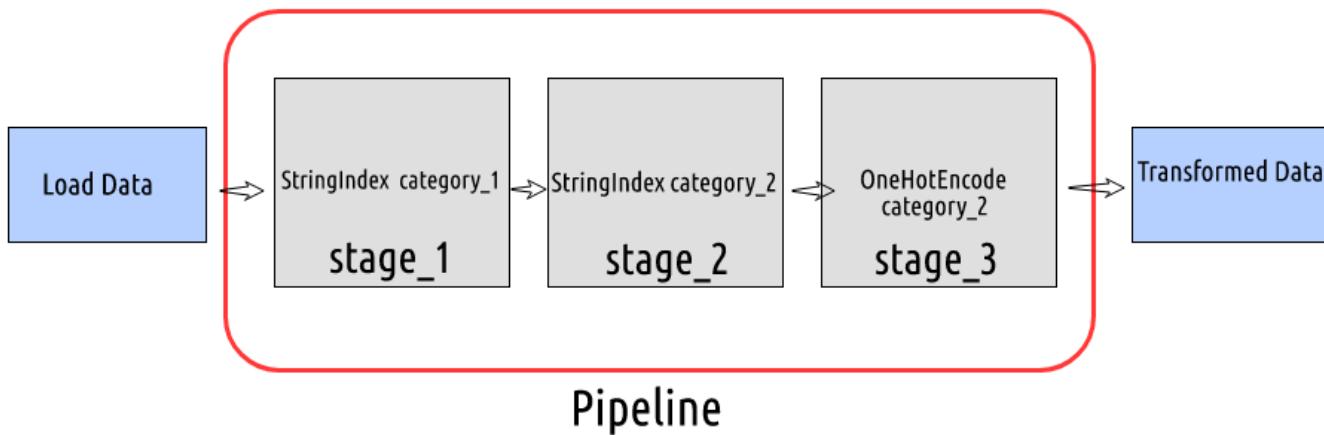
```
from pyspark.ml import Pipeline

# Crearea unui dataframe exemplu
sample_df = spark.createDataFrame([
    (1, 'L101', 'R'),
    (2, 'L201', 'C'),
    (3, 'D111', 'R'),
    (4, 'F210', 'R'),
    (5, 'D110', 'C')
], ['id', 'category_1', 'category_2'])

sample_df.show()
```

Etapa 5 – Construirea *ML Pipeline*

- Transformăm datele în ordinea următoare:
 - Etapa 1: *String Index* asupra coloanei *category_1*
 - Etapa 2: *String Index* asupra coloanei *category_2*
 - Etapa 3: *One-Hot Encode* asupra coloanei indexate *category_2*



Etapa 5 – Construirea *ML Pipeline*

- La fiecare etapă vom transmite numele coloanelor de intrare și de ieșire și vom configura *pipeline-ul* transmitând etapele definite ca argument de tip listă al constructorului *Pipeline*.
- Modelul de *pipeline* obținut va efectua anumiți pași unul câte unul, secvențial, și va conduce către rezultatul final.
- Implementarea *pipeline-ului*:

```
# definirea etapei 1: transformam coloana category_1 in tip numeric
stage_1 = StringIndexer(inputCol= 'category_1', outputCol= 'category_1_index')
# definirea etapei 2: transformam coloana category_2 in tip numeric
stage_2 = StringIndexer(inputCol= 'category_2', outputCol= 'category_2_index')
# definirea etapei 3 : one hot encoding asupra coloanei numerice category_2
stage_3 = OneHotEncoderEstimator(inputCols=['category_2_index'], outputCols=['category_2_OHE'])

# crearea pipeline-ului
pipeline = Pipeline(stages=[stage_1, stage_2, stage_3])

# aplicarea functiei fit asupra pipeline-ului si transformarea datelor conform acestuia
pipeline_model = pipeline.fit(sample_df)
sample_df_updated = pipeline_model.transform(sample_df)

# vizualizarea datelor afisate
sample_df_updated.show()
```

Etapa 5 – Construirea *ML Pipeline*

- **Exemplul 2**

- Transformări asupra datelor și construirea unui model de regresie logistică
- Crearea unui dataframe ce va fi setul de date de training, cu 4 caracteristici și o etichetă întă:

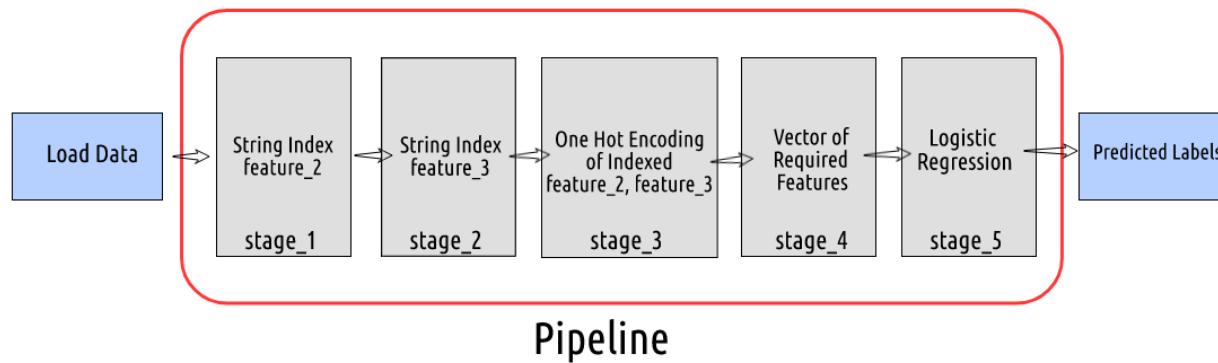
```
from pyspark.ml.classification import LogisticRegression

# crearea dataframe-ului - 4 caracteristici și o coloana eticheta
sample_data_train = spark.createDataFrame([
    (2.0, 'A', 'S10', 40, 1.0),
    (1.0, 'X', 'E10', 25, 1.0),
    (4.0, 'X', 'S20', 10, 0.0),
    (3.0, 'Z', 'S10', 20, 0.0),
    (4.0, 'A', 'E10', 30, 1.0),
    (2.0, 'Z', 'S10', 40, 0.0),
    (5.0, 'X', 'D10', 10, 1.0),
], ['feature_1', 'feature_2', 'feature_3', 'feature_4', 'label'])

# afisarea datelor
sample_data_train.show()
```

Etapa 5 – Construirea *ML Pipeline*

- Presupunem că ordinea acțiunilor din *pipeline* este aceasta:
 - Etapa 1: *String Index* asupra coloanei *feature_2*
 - Etapa 2: *String Index* asupra coloanei *feature_3*
 - Etapa 3: *One Hot Encoding* asupra coloanei indexate corespunzătoare lui *feature_2* și *feature_3*
 - Etapa 4: Crearea unui vector al tuturor caracteristicilor necesare pentru a antrena un model de Regresie logistică
 - Etapa_5: Construirea modelului de regresie logistică



Etapa 5 – Construirea *ML Pipeline*

- Definim etapele specificând numele coloanelor de intrare și ale celor de ieșire
- Etapa finală constă în construirea modelului de regresie logistică
- La final, când vom executa *pipeline*-ul pe setul de date de *training*, acesta va rula pașii din secvență și va adăuga coloane noi în dataframe (precum *rawPrediction*, *probability*, *prediction*)

Etapa 5 – Construirea *ML Pipeline*

```
# definirea etapei 1: transformarea coloanei feature_2 in tip numeric
stage_1 = StringIndexer(inputCol= 'feature_2', outputCol= 'feature_2_index')
# definirea etapei 2: transformarea coloanei feature_3 in tip numeric
stage_2 = StringIndexer(inputCol= 'feature_3', outputCol= 'feature_3_index')
# definirea etapei 3: one hot encoding asupra valorilor numerice corespunzatoare coloanelor feature_2 si
feature_3, generate in etapele 1 si 2
stage_3 = OneHotEncoderEstimator(inputCols=[stage_1.getOutputCol(), stage_2.getOutputCol()],
                                  outputCols= ['feature_2_encoded', 'feature_3_encoded'])
# definirea etapei 4: crearea unui vector al tuturor caracteristicilor necesare pentru a antrena modelul
de regresie logistica
stage_4 = VectorAssembler(inputCols=['feature_1', 'feature_2_encoded', 'feature_3_encoded',
                                     'feature_4'], outputCol='features')
# definirea etapei 5: modelul de regresie logistica
stage_5 = LogisticRegression(featuresCol='features', labelCol='label')

# crearea pipeline-ului
regression_pipeline = Pipeline(stages= [stage_1, stage_2, stage_3, stage_4, stage_5])

# antrenarea modelului pe datele de antrenament
model = regression_pipeline.fit(sample_data_train)
# transformarea datelor
sample_data_train = model.transform(sample_data_train)

# afisarea catorva dintre coloanele generate
sample_data_train.select('features', 'label', 'rawPrediction', 'probability', 'prediction').show()
```

Etapa 5 – Construirea *ML Pipeline*

- Se creează un set de date de test fără etichete. Nu mai este necesar să definim din nou pașii, ci doar vom transmite aceste date *pipeline*-ului :

```
# crearea unui set de date fara etichete
sample_data_test = spark.createDataFrame([
    (3.0, 'Z', 'S10', 40),
    (1.0, 'X', 'E10', 20),
    (4.0, 'A', 'S20', 10),
    (3.0, 'A', 'S10', 20),
    (4.0, 'X', 'D10', 30),
    (1.0, 'Z', 'E10', 20),
    (4.0, 'A', 'S10', 30),
], ['feature_1', 'feature_2', 'feature_3', 'feature_4'])

# transformarea datelor utilizand pipeline-ul
sample_data_test = model.transform(sample_data_test)

# afisarea valorilor de predictie obtinute asupra datelor de test
sample_data_test.select('features', 'rawPrediction', 'probability',
'prediction').show()
```

Concepțe *Pipelines*

- MLlib standardizează API-urile pentru algoritmii ML cu scopul de a simplifica combinarea mai multor algoritmi într-un singur *pipeline* (sau workflow)
- Concepțele cheie ale Pipelines API (unde conceptul de *pipeline* este inspirat din proiectul scikit-learn):
 - DataFrame
 - *Transformer*: Algoritm ce poate transforma un DataFrame în alt DataFrame
 - Un model ML este un Transformer ce transformă un dataframe cu caracteristici într-un dataframe cu predicții
 - *Estimator*: Algoritm ce poate fi antrenat pe un DataFrame pentru a produce un Transformer
 - Un algoritm de învățare este un estimator ce se antrenează pe un DataFrame și produce un model
 - *Pipeline*: Un *pipeline* înlănțuie mai multe obiecte *Transformer* și *Estimator*, specificând astfel un workflow
 - *Parameter*: Obiectele *Transformer* și *Estimator* au un API comun pentru specificarea parametrilor

Transformer

- Un *Transformer* este o abstractizare ce include transformări de caracteristici și modele învățate
- Implementează o metodă *transform*, ce convertește un DataFrame în altul, în general prin adăugarea uneia sau mai multor coloane
- Exemple:
 - Un transformator de caracteristici poate lua un DataFrame, din care citește o coloană (text), pe care o mapează în altă coloană (de exemplu, vector de caracteristici) și returnează un nou DataFrame în care a fost adăugată coloana mapată
 - Un model de învățare poate lua un DataFrame, poate citi coloana ce conține vectorii de caracteristici, poate face predicții asupra etichetei corespunzătoare fiecărui vector de caracteristici și returnează un nou DataFrame cu etichetele de predicție adăugate ca o nouă coloană.

Estimator

- Un *Estimator* abstractizează conceptul de algoritm de învățare sau, în general, cel de algoritm care ajustează (*fit*) sau antrenează date.
 - Un estimator implementează o metodă *fit*, care acceptă un *DataFrame* și produce un *Model*, care este un *Transformer*.
 - De exemplu, un algoritm de învățare precum *LogisticRegression* este un *Estimator*, iar apelul metodei *fit()* antrenează un *LogisticRegressionModel*, care este un *Model* și deci un *Transformer*.
- Fiecare instanță de *Transformer* sau *Estimator* are un ID unic, care va fi folositor în specificarea parametrilor.

Parametri

- Există un API uniform pentru specificarea parametrilor pentru *Estimator* și *Transformer*.
- Un *Param* este un parametru cu nume
- Un *ParamMap* este o mulțime de perechi (parametru, valoare)
- Există 2 modalități principale de a transmite parametri unui algoritm:
 1. Se setează parametrii pentru instanță. De exemplu, dacă *lr* este o instanță a lui *LogisticRegression*, se poate apela *lr.setMaxIter(15)* pentru a determina *lr* să folosească cel mult 15 iterării
 2. Se transmite un *ParamMap* lui *fit()* sau *transform()*. Parametrii din *ParamMap* vor suprascrie parametrii specificați anterior prin metodele *setter*.

Parametri

- Parametrii aparțin instanțelor particulare de *Estimator* și *Transformer*. De exemplu, dacă avem două instanțe *lr1* și *lr2* ale lui *LogisticRegression*, se poate construi un *ParamMap* cu ambele valori ale parametrilor *maxIter* specificate: *ParamMap(lr1.maxIter -> 10, lr2.maxIter -> 20)*
 - Acest lucru este util dacă avem doi algoritmi cu parametrul *maxIter* în *Pipeline*

Demo Spark

- Demo Estimatori, Transformatori și Parametri:
https://github.com/apache/spark/blob/master/examples/src/main/python/ml/estimator_transformer_param_example.py
- Demo Pipeline:
https://github.com/apache/spark/blob/master/examples/src/main/python/ml/pipeline_example.py

Selectia modelelor (*hyperparameter tuning*)

- Un beneficiu al utilizării *Pipeline*-urilor este optimizarea hiperparametrilor
- Selectarea modelului se poate realiza în mod automat
- Tool-uri precum *CrossValidation* permit optimizarea hiperparametrilor atât din algoritmi individuali, cât și din *pipeline*-uri.
- Selectarea modelului este un *task* important din ML
- Selectarea modelului presupune utilizarea datelor pentru a găsi cel mai bun model sau cei mai buni parametri pentru un *task* dat.
 - Mai poate numele de *tuning*.
- *Tuning*-ul se poate realiza fie pentru Estimatori individuali precum *LogisticRegression* sau pentru *Pipeline*-uri întregi ce includ mai mulți algoritmi, prelucrare de caracteristici și alți pași.
 - Utilizatorul poate optimiza un întreg *pipeline* odată, nu doar fiecare element separat al acestuia.

Selectia modelelor

- MLlib realizează selecția modelelor prin intermediul unor *tool-uri* precum *CrossValidator* și *TrainValidationSplit*.
- Acestea necesită următoarele elemente:
 - *Estimator*: algoritm sau *pipeline* de optimizat
 - Multime de obiecte *ParamMap*: parametri din care se poate alege (denumit și *grid* de parametri)
 - *Evaluator*: metrică ce măsoară cât de bine se comportă un *Model* antrenat pe date de test

Selectia modelelor

- Aceste *tool-uri* de selecție a modelelor funcționează astfel:
 - Divizează datele de intrare în seturi de date de *training* și de test.
 - Pentru fiecare pereche (*training*, test), iterează prin mulțimea de obiecte *ParamMap*:
 - Pentru fiecare *ParamMap*, antrenează estimatorul folosind acei parametri, obțin modelul antrenat și evaluează performanța lui folosind evaluatorul.
 - Selectează modelul produs de setul de parametri cu cele mai bune rezultate.
- Evaluatorul poate fi de tip:
 - *RegressionEvaluator* pentru problemele de regresie
 - *BinaryClassificationEvaluator* pentru datele binare
 - *MulticlassClassificationEvaluator* pentru problemele multi-clasă
 - *MultilabelClassificationEvaluator* pentru clasificările multi-label
 - *RankingEvaluator* pentru problemele de *ranking*.
- Metrica implicită utilizată pentru alegerea celui mai bun *ParamMap* poate fi suprascrisă de către metoda *setMetricName* în fiecare dintre acești evaluatori.

Selectia modelelor

- Construirea *grid*-ului de parametri se realizează cu ajutorul *ParamGridBuilder*.
- Implicit, seturile de parametri din *grid* se evaluatează secvențial.
 - Se poate modifica acest comportament, setând o valoare >1 parametrului *parallelism* înainte de a lansa în execuție selectarea modelului (cu *CrossValidator* sau *TrainValidationSplit*).

Cross Validation

- *CrossValidator* împarte setul de date într-o mulțime de diviziuni ce vor fi folosite ca seturi de date separate de *training* și de test.
 - De exemplu, pentru $k=3$ diviziuni, *CrossValidator* va genera 3 perechi de *dataset*-uri (*training*, *test*), fiecare dintre acestea folosind 2/3 din date pentru antrenare și 1/3 pentru testare.
- Pentru a evalua un *ParamMap*, *CrossValidator* calculează media metricii de evaluare pentru cele 3 modele produse prin antrenarea estimatorului pe cele 3 perechi diferite de *dataset*-uri (*training*, *test*)
- După identificarea celui mai bun *ParamMap*, *CrossValidator* re-antrenează estimatorul folosind cel mai bun *ParamMap* și setul de date întreg.
- Observație: *CrossValidation* peste un grid de parametri este o acțiune costisitoare.

Demo Spark

- Demo Cross Validation:
https://github.com/apache/spark/blob/master/examples/src/main/python/ml/cross_validator.py
 - Exemplul are un grid de parametri cu 3 valori pentru *hashingTF*, *numfeatures*, 2 valori pentru *lr.regParam* iar *CrossValidator* utilizează 2 diviziuni => 12 modele diferite ce vor fi antrenate
 - În realitate, pot fi luați în considerare mai mulți parametri și mai multe diviziuni (valorile $k=3$ și $k=10$ fiind întâlnite în mod obișnuit).
 - *CrossValidator* este costisitor, dar rămâne o bună metodă de alegere a parametrilor, care este mai robustă din punct de vedere statistic decât *tuning*-ul heuristic, manual.

Train-Validation Split

- O altă metodă de *tuning* al hiperparametrilor oferită de Spark.
- *TrainValidationSplit* evaluează fiecare combinație de parametri o singură dată, spre deosebire de *CrossValidator* (care evaluează de k ori)
- Este mai puțin costisitor, dar nu va produce rezultate atât de bune atunci când setul de date de *training* nu este suficient de mare
- Creează o singură pereche de *dataset*-uri (*training*, *test*). Divizează *dataset*-ul în aceste 2 părți cu ajutorul parametrului *trainRatio*.
 - Pentru *trainRatio*=0.75 => generează o pereche de *dataset*-uri în care 75% din date sunt folosite pentru *training* și 25% pentru validare.
- Similar *CrossValidator*, la final *TrainValidationSplit* antrenează estimatorul folosind cel mai bun *ParamMap* și întregul *dataset*.

Demo Spark

- Demo *TrainValidationSplit*:
https://github.com/apache/spark/blob/master/examples/src/main/python/ml/train_validation_split.py

Bibliografie

- <https://spark.apache.org/docs/latest/ml-guide.html>
- [https://indico.cern.ch/event/761215/contributions/3158971/attachm
ents/1724310/2784714/SparkPipeline.pdf](https://indico.cern.ch/event/761215/contributions/3158971/attachments/1724310/2784714/SparkPipeline.pdf)
- [https://www.analyticsvidhya.com/blog/2019/11/build-machine-
learning-pipelines-pyspark/](https://www.analyticsvidhya.com/blog/2019/11/build-machine-
learning-pipelines-pyspark/)