Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Spark SQL és Spark ML

Összefoglalás

Az olvasó ebből az olvasóleckéből mélyebb elméleti ismereteket szerezhet az Apache Spark két fontos moduljáról, a Spark SQL-ről, illetve a Spark ML-ről. Előbbi esetében áttekintést adunk a Spark strukturált adatok lekérdezéséhez nyújtott támogatásáról, az alkalmazott adat absztrakciókról, illetve az egyes programozói interfészekről. Rövid áttekintést kap az olvasó a gépi tanulási feladatról általában, annak típusairól és legygakrabban használt algoritmusairól. Végezetül a Spark gépi tanuló moduljának koncepcionális működését ismertetjük az olvasóval, ami után képes lesz akár önállóan is gépi tanuló folyamatok összeállítására.
A lecke fejezetei:

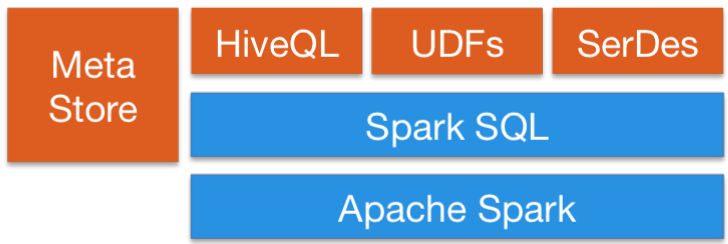
- 1. fejezet: **A Spark SQL bemutatása, kapcsolódása más technológiákhoz (olvasó)**
- 2. fejezet: **Rövid áttekintő a gépi tanulás által megoldható problémákba, algoritmusok kategorizálása (olvasó)**
- 3. fejezet: **A Spark ML (MLlib) gépi tanuló modul koncepcionális bemutatása példa kóddal együtt (olvasó)**

Téma típusa: **elméleti**
Olvasási idő: **55 perc**

1. fejezet

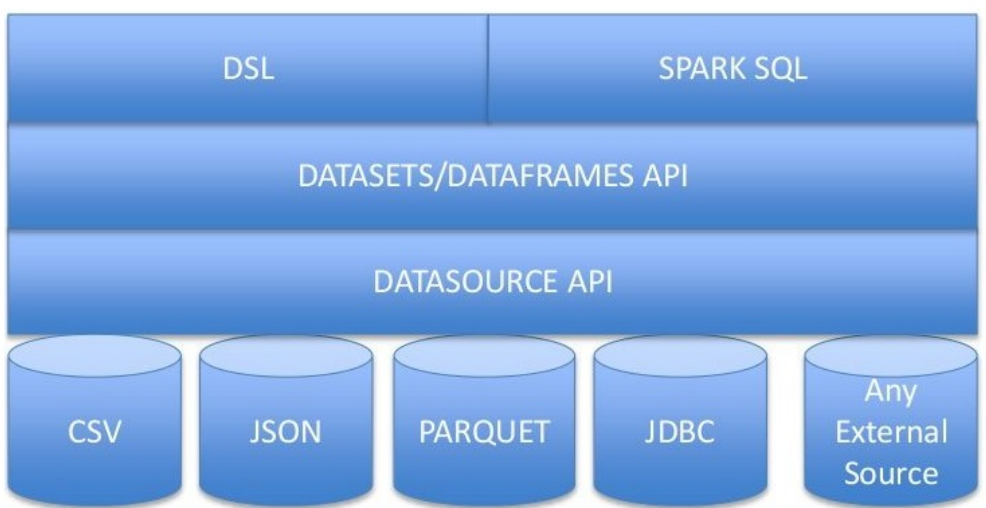
Strukturált adat lekérdezés Spark fölött a Spark SQL segítségével

A Spark részét képezi a Spark SQL modul [1], ami a strukturált adatfeldolgozást támogatja. Funkcióját tekintve megegyezik az Apache Hive-val [2], ám bizonyos Hive hiányosságokat (mint például a MapReduce job-ok gyengébb hatékonyságát kis és közepes adathalmazokon vagy hogy a Hive a sikertelen lekérdezéseket nem tudja onnan folytatni, ahol hiba folytán leállt, hanem mindig a teljes lekérdezést újrafuttatja) kiküszöböl, és mint ilyen a Hive utódjaként tekinthető a Spark stack-en belül. Ez nem azt jelenti, hogy a Spark SQL konkurrens technológiák és vagy az egyiket használjuk vagy a másikat. Teljes körű az integráció és együttműködés a két eszköz között. Például a Spark SQL támogatja az adatok betöltését Hive táblákból [3] és a HiveQL lekérdező nyelvet is (lásd ábra), illetve maga az Apache Hive is tudja használni a Spark motort a lekérdezések végrehajtásához, nem csak a klasszikus MapReduce-t (sőt, az újabb Hive verziók már elavultnak jelölték a MapReduce használatát, és a későbbi verziókban meg is fog szűnni a támogatása).



> https://spark.apache.org/images/sql-hive-arch.png

A Spark SQL integrálja a relációs adat feldolgozást a Spark funkcionális programozási modelljével. Több féle adatforrást is támogat, és az SQL-ben megfogalmazott lekérdezéseket Spark transzformációkká és akciókká alakítja. A klasszikus RDD API-hoz képest a Spark SQL sokkal több információt tartalmaz mind az adatok, mind pedig a rajtuk végzendő műveletek struktúrájáról, amit a Spark végrehajtáskor hatékonyan ki tud használni a végrehajtás optimalizálásához. A a Spark elég jól elmossa a határvonalat az RDD-k, és a relációs táblák között, azáltal hogy az integrációhoz a már korább olvasóleckékben ismertetett DataFrame API-t használja (lásd lenti ábra). Ez nagyobb optimalizálási lehetőséget ad a Spark kezébe, ezért a DataFrame/Dataset API a preferált mód a Spark SQL eléréséhez (noha lehetőség van közvetlen SQL végrehajtásra is).

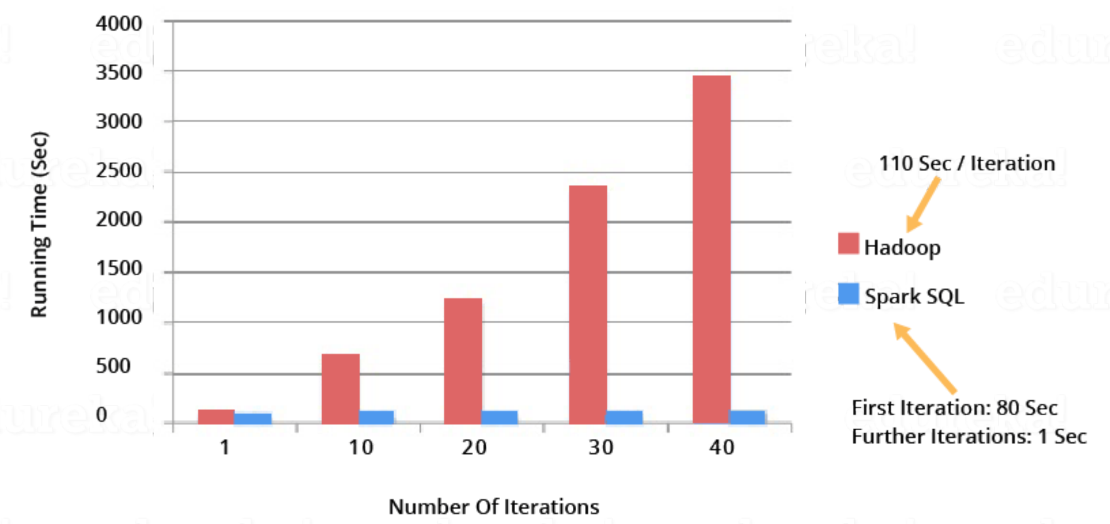


> https://i.pinimg.com/originals/c4/11/2b/c4112bb4bb2c8b7e61ce1425a0d10051.jpg

A Spark SQL fő rétegei

A Spark SQL az alábbi négy osztály könyvtárat használja a relációs és procedurális adatfeldolgozáshoz:

1. **Data Source API:** egy univerzális API strukturált adatok betöltéséhez és tárolásához:
    - Beépített támogatás az Avro, CSV, Elasticsearch, JSON, JDBC, Parquet, Cassandra, stb. adatformátumokhoz és a HDFS, Hive, MySQL, stb. tároló rendszerekhez.
    - A Spark-core modul segítségével könnyen integrálható tetszőleges más BigData platformmal.
    - Python, Java, Scala és R API támogatása.
2. **DataFrame API:** a Spark RDD fölötti absztrakció, adatok elosztottan tárolt gyűjteményének, amelyek névvel ellátott oszlopokba vannak szervezve. Felépítését tekintve megegyezik egy relációs adattáblával SQL esetén (a DataFrame-eket részletesen bemutattuk a `9e_BigData-spark-rdd-df-SPOC` olvasóleckében).
3. **SQL interpreter és optimalizáló:** egy funkcionális programnyelven (Scala) írt modul, amely:
    - A Spark SQL legfejlettebb és legújabb komponense.
    - Egy általános keretrendszert ad fák transzformációjához, amely lekérdezések elemzéséhez/kiértékeléséhez, optimalizálásához, futtatási terv készítéséhez, stb.
    - Lehetővé teszi a költség alapú és szabály alapú optimalizálást (pl. Catalyst [4]), amelyek hatására az SQL lekérdezések sokkal gyorsabban lefutnak, mint az RDD fölötti műveletek. A Spark SQL és Hadoop teljesítmény összehasonlítást az alábbi ábra szemlélteti (*forrás:* edureka!).



> https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2016/12/Performance-Spark-SQL-Vs-Hadoop-Spark-SQL-Edureka.png

4. **SQL szolgáltatás:** ez a komponens a kiindulópontja a Spark strukturált adatokkal történő munkának. Ez lehetővé teszi DataFrame-ek létrehozását, valamint SQL lekérdezések végrehajtását.

Az alábbi Java kódrészlet a Spark SQL használatát szemlélteti (további részletekért lásd a Spark programozási útmutatót [5]):

```
 1. import org.apache.spark.sql.SparkSession;
 2.
 3. public class SparkSQLExample {
 4.
 5. public static void main(String[] args) {
 6.         SparkSession spark = SparkSession
 7.           .builder()
 8.           .appName("Java Spark SQL basic example")
 9.           .config("spark.some.config.option", "some-value")
10.           .getOrCreate();
11.
12.         Dataset<Row> df =
spark.read().json("examples/src/main/resources/people.json");
13.
14.         // Displays the content of the DataFrame to stdout
15.         df.show();
16.         // +----+-------+
17.         // | age|   name|
18.         // +----+-------+
19.         // |null|Michael|
20.         // |  30|   Andy|
21.         // |  19| Justin|
22.         // +----+-------+
23.
24.         // Register the DataFrame as a SQL temporary view
25.         df.createOrReplaceTempView("people");
26.
27.         Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
28.         sqlDF.show();
29.         // +----+-------+
```

```
30.         // | age|   name|
31.         // +----+-------+
32.         // |null|Michael|
33.         // |  30|   Andy|
34.         // |  19| Justin|
35.         // +----+-------+
36.     }
37. }
```

A 6. sorban létrehozunk egy `SparkSession` objektumot, ami a megfelelő Spark kontextust reprezentálja (Spark és egyéb komponensek paraméterei, stb.). Ennek segítségével a 12. sorban létre tudunk hozni egy DataFrame-t a `people.json` fájl beolvasásával. A DataFrame-n közvetlenül is végezhetünk műveleteket (lásd 15. sor), de ahogy a 25. sorban látszik, a DataFrame-et egy ideiglenes SQL táblává is tehetjük, hogy aztán a Spark SQL segítségével (`spark.sql` hívás) közvetlenül SQL-el kérdezhessük le az adatokat.

📖 2. fejezet

Gépi tanulás bevezető

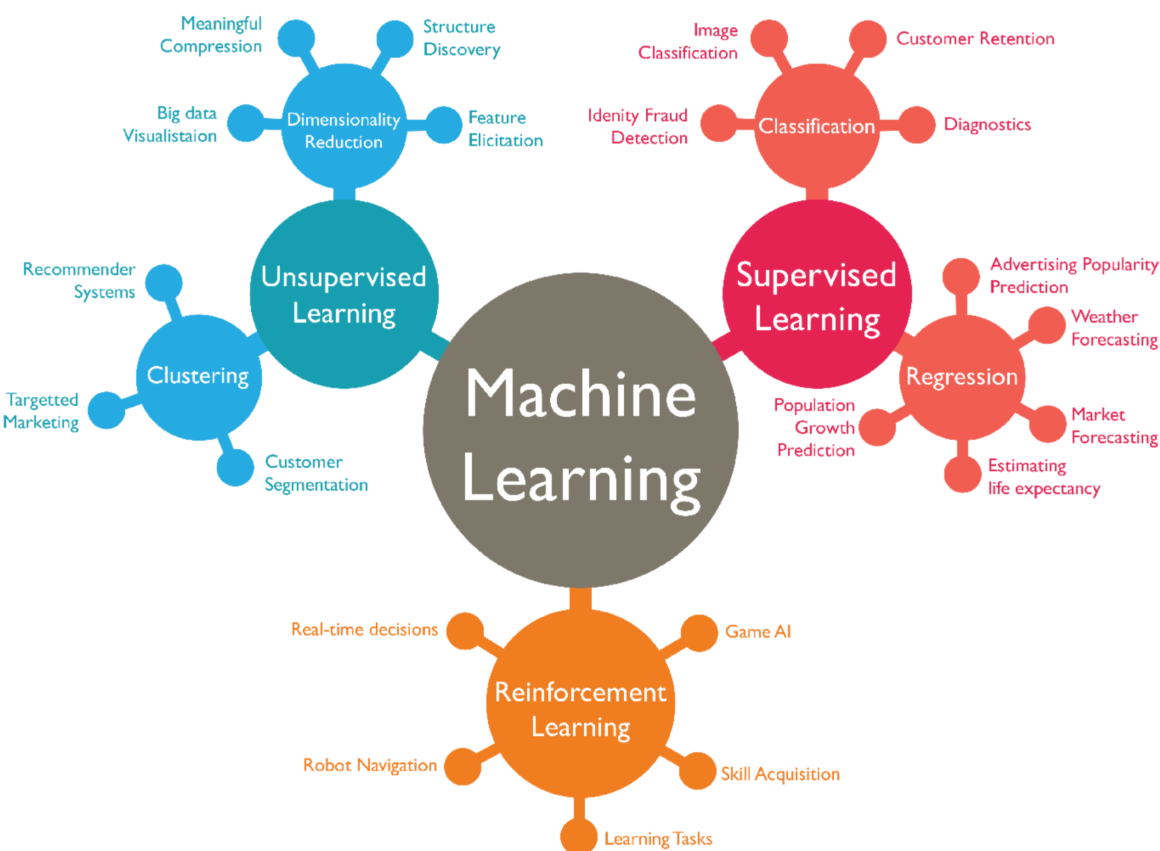
A gépi tanulási módszerek igen jelentős szerepet kapnak napjaink problémáinak megoldásában. Természetesen BigData feldolgozásához is számos felhasználási területe van. A gépi tanulás a mesterséges intelligencia egyik alterülete, ahova olyan statisztikai módszereket sorolunk, amelyek képesek véges számú ún. tanuló mintából egy olyan függvényt előállítani, mellyel később még sosem látott mintához is előállít nekünk bizonyos információt (amit megtanult). Tipikus gépi tanulási problémák:

- Karakter felismerés
- Arcfelismerés
- Beszédfelismerés/szintetizálás
- Spam szűrés
- Szentiment analízis
- Hiba előrejelzés

Számos algoritmus létezik, amelyek különböző módszerekkel valósítják meg a "tanulást" (függvény illesztést). Ezeket az algoritmusokat alapvetően két csoportba sorolhatjuk:

- **Felügyelt tanulás (supervised learning)** - a tanulás során rendelkezésre állnak címkézett adatok (általában kézzel meghatározott információ, amik az egyes tanuló példákhoz vannak rendelve, pl. képen látható objektumok megnevezése, egy e-mail spam-e, stb.). A felügyelt tanuló módszereket az osztály címkék milyensége alapján két nagyobb csoportra bonthatjuk:
  - *Osztályozás (classification)*: a tanuló adatokhoz diszkrét címkék vannak megadva, és a feladat a minták bizonyos jellemzői alapján ezeket a konkrét címkéket megtanulni (pl. adott egy betűt ábrázoló kép, döntsük el milyen betűt ábrázol). Az osztálycímkék számossága alapján megkülönböztetünk bináris (csak két fajta címke van, pozitív és negatív) és multi label (több különböző címke közül választhat a modell) osztályozást.
  - *Regresszió (regression)*: itt a címkék folytonosak, és a modelltől is azt várjuk, hogy egy újabb mintához egy folytonos értéket adjon eredményül, ne pedig egy konkrét címkét. A regressziós módszerek könnyen osztályozó módszerekké alakíthatók (egy vagy több határértéket kell definiálni, amely mentén diszkrét címkéket rendelhetünk a modell predikciójához)

- **Nem felügyelt tanulás (unsupervised learning)** - ebben az esetben a tanuláshoz nem áll rendelkezésre címkézett/kézzel előállított tanuló adathalmaz. Ebben az esetben az ún. klaszterezési eljárásokat alkalmazhatjuk, amelyek az egymáshoz valamilyen szempontból hasonló mintákat próbálják meg összecsoportosítani. Az új minta érkezésekor ez alapján el tudjuk dönteni, hogy melyik már meglévő csoporthoz hasonlít leginkább. A klaszterezés mellett az olyan módszereket is ide soroljuk, amelyek a bemenő adatok jellemző terének dimenzióját csökkentik a redundáns jellemzők elhagyásával.
- **Megerősítő tanulás (reinforcement learning)** - egy speciális gépi tanuló módszer, ahol a modell döntések sorozatát hozza meg. Nincs adatcímkézés (a nem felügyelt módszerekhez hasonlít), a modell az alapján tanul, hogy bizonyos döntéseiért jutalom vagy büntetés jár (azaz folyamatos visszajelzés/megerősítés alapján tanul), így próbál egy komplex helyzetben megfelelő döntéseket hozni (pl. játék AI, robot navigáció).



> https://www.normshield.com/wp-content/uploads/2017/01/MachineLearningDiagram.png

Az összes gépi tanuló módszer valamilyen jellemzők ún. prediktorok (feature) alapján próbálja megtanulni a mintához tartozó címkét. Például egy karakter felismerési feladatnál egy prediktor lehet a képen szereplő fekete és fehér képpontok egymáshoz viszonyított arány, stb. Ha egy táblába rendezzük az egyes mintákat úgy, hogy minden minta egy sor, és az oszlopok pedig az adott mintához tartozó prediktorok értéke, illetve a megtanulni kívánt osztály címkéje, egy klasszikus tanuló táblát kapunk. A legtöbb tanuló algoritmus ilyen bemeneten dolgozik (amelyek tipikusan BigData környezetben is rendelkezésre állnak). A tanulás tipikus módja, hogy ezt a tanuló adathalmazt 3 részre osztjuk:

- **Tanuló (train) adatok** - a rendelkezésre álló minta halmazból különválasztott legnagyobb (tipikusan 80-90%) részhalmaz, amin gyakorlatilag a modellek betanítását el tudjuk végezni.
- **Optimalizálásra szolgáló (dev) adatok** - a tanuló algoritmusok működését számos paraméter befolyásolja, a megfelelő értékek beállítását ezen az elkülönített kisebb (kb. 5-10%) részhalmazon végzik
- **Tesztelési (test) adatok** - a betanított modell validálására, teljesítményének meghatározására szolgáló adat részhalmaz (tipikusan 10-20%). A teszteléshez szintén

ismernünk kell a címkéket, hogy el tudjuk dönteni a modell az esetek mekkora hányadában ad helyes eredményt. Egy modell teljesítményét sok statisztikai mérőszámmal jellemezhetjük, pl. accuracy, precision, recall, F-measure, loss, ROC, AUC, korreláció, MCC [6].

A leggyakrabban használt gépi tanuló algoritmusokat a következő URL jól összefoglalja: https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/

3. fejezet

Gépi tanulás Spark fölött, a Spark ML (MLlib) modul bemutatása

Az MLlib [7] (vagy újabban Spark ML) modul a Spark gépi tanulást támogató komponense. Segít abban, hogy a Hadoop klaszteren tárolt adatainkra gyorsan, könnyedén és jól skálázható módon alkalmazhassunk gépi tanuló algoritmusokat. Az MLlib az újabb verzióktól kezdve a DataFrame API-ra épül, noha az RDD alapú API is karbantartás alatt marad. Magas szinten az MLlib a következő funkcionalitást nyújtja:

- Gépi tanuló algoritmus: a leggyakoribb osztályozó, regressziós, klaszterező módszerek implementációi
- Feature kezelés: feature kinyerés, transzformáció, dimenzió csökkentés és feature szelekció
- Csővezetékek: eszközök gépi tanuló pipeline-ok létrehozásához, kiértékeléséhez és optimalizálásához
- Tárolás: algoritmusok, modellek és pipeline-ok mentése és visszatöltése
- Egyéb kiegészítő eszközök: lineáris algebra, statisztika, adat kezelés, stb.

MLlib csővezeték fogalmak

Az MLlib egységesíti és standardizálja a gépi tanuláshoz használható API-kat, ezzel megkönnyítve több algoritmus egyetlen ún. csővezetékbe (pipeline/workflow) szervezését. Gépi tanulás során nagyon gyakori, hogy egy modell tanítás több algoritmus egymás után történő végrehajtásával áll elő (adat előkészítés, feature kinyerés, modell tanítás, stb.), az MLlib ilyen pipeline-ok felállítását könnyíti meg az alábbi absztrakciók felhasználásával:

- `DataFrame` - az ML API a DataFrame-eket használja tanuló adathalmaznak (lásd tanuló tábla az előző fejezetből). A DataFrame-ek különböző típusú adatokat tartalmazhatnak, pl. különböző oszlopokban tartalmazhat szöveget, feature vektorokat, pozitív osztály címkéket, vagy predikciókat.
- `Transformer` - olyan algoritmus, amely egy `DataFrame`-et egy másik `DataFrame`-é tud alakítani. Például egy gépi tanuló modell egy `Transformer`, ami egy feature-öket tartalmazó `DataFrame`-et tud átalakítani egy predikciókat tartalmazó `DataFrame`-mé.
- `Estimator` - olyan algoritmus, amely egy `DataFrame` adataira illesztve egy `Transformer`-t állít elő. Például egy gépi tanuló algoritmus egy `Estimator`, amely egy `DataFrame`-en tanul és egy modellt állít elő (ami ugye egy `Transformer`).
- `Pipeline` - egy csővezeték több `Transformer`-t és `Estimator`-t tud összefűzni, ezáltal egy teljes gépi tanuló folyamatot leírni.
- `Parameter` - az összes `Transformer` és `Estimator` egy közös API-t használ a paramétereinek leírásához.

Transformer-ek

A `Transformer`-ek absztrakció magában foglalja a feature transzformációkat és a betanított modelleket. Technikailag egy `Transformer` egy `transform()` metódust implementál, amely egy `DataFrame`-ből egy másik `DataFrame`-et állít elő, tipikusan egy vagy több oszlop hozzáadásával. Például:

- Egy feature `Transformer` vehet egy `DataFrame`-et, kiolvas belőle egy oszlopot (pl. "text"), amihez előállít egy új oszlopot (pl. feature vektorokat), és kiír egy `DataFrame`-et az új oszlop hozzáadásával.
- Egy tanuló modell vehet egy `DataFrame`-et, aminek beolvassa a feature vektor oszlopait, elvégzi a predikciót és kiír egy új `DataFrame`-et a prediktált osztálycímkékkel kiegészítve.

Estimator-ok

Az `Estimator` a klasszikus tanuló algoritmusok általánosítása, vagy bármely olyan algoritmusé, amely tanulást vagy illesztést végez adatokon. Technikailag egy`Estimator` egy `fit()` metódust implementál, ami egy `DataFrame`-et fogad és egy `Model`-t állít elő, ami egy `Transformer`. Például a `LogisticRegression` tanuló algoritmus egy `Estimator`, aminek a `fit()` metódusa betanít (előállít és visszaad) egy `LogisticRegressionModel`-t, ami egy `Model`, ezáltal egy `Transformer` is.

Parameter-ek

Mind a `Transformer`-ek mind az `Estimator`-ok egy közös API-t, a `Parameter`-t használják a paramétereik leírásához. A`Param` egy névvel ellátott paraméter. A `ParamMap` pedig egy `(paraméter, érték)` párokból álló halmaz. Az algoritmusokat két módon paraméterezhetjük:

1. Beállíthatjuk közvetlenül egy példányon, pl. ha az `lr` egy példány a `LogisticRegression` algoritmusból, meghívhatjuk rajta a `lr.setMaxIter(10)` beállítást, hogy az `lr.fit()` maximum 10 iterációt használjon.
2. Átadhatunk egy `ParamMap`-et a `fit()` vagy `transform()` metódusnak. A `ParamMap`-ben lévő paraméterek felülírnak bármely olyan paramétert, amit korábban a setter metódussal állítottunk be.

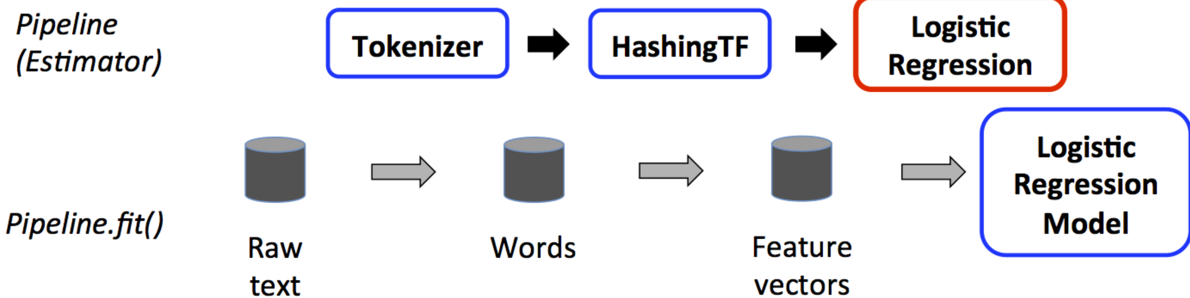
ML pipeline működése

Egy `Pipeline` fázisok sorozata, ahol minden egyes fázis vagy egy `Transform` vagy egy `Estimator`. Ezek a fázisok egymás után sorba rendezve kerülnek futtatásra, és a bemenő `DataFrame` minden fázis során módosul. Egy `Transformer` fázis során annak `transform()` metódusa hívódik meg a `DataFrame`-re. Az `Estimator` fázisok esetén a `fit()` metódus hívódik meg, ami előállít egy `Transform`-ot (ami ezek után a `PipelineModel` vagy más néven illesztett `Pipeline` részévé válik), és ennek a `Transform`-nak a `transform()` metódusa kerül meghívásra a `DataFrame`-en.

Vegyük az alábbi egyszerű szöveges dokumentum feldolgozó tanulási folyamatot, amin aztán a fent leírt pipeline koncepciót illusztráljuk:

- Bontsuk minden dokumentum szövegét szavakra.
- Konvertáljuk át minden dokumentum szavait számszerű feature vektorokká.
- Tanítsunk be egy modellt a feature vektorok és osztálycímkék alapján.

A `Pipeline` tanítás idejű felhasználása.

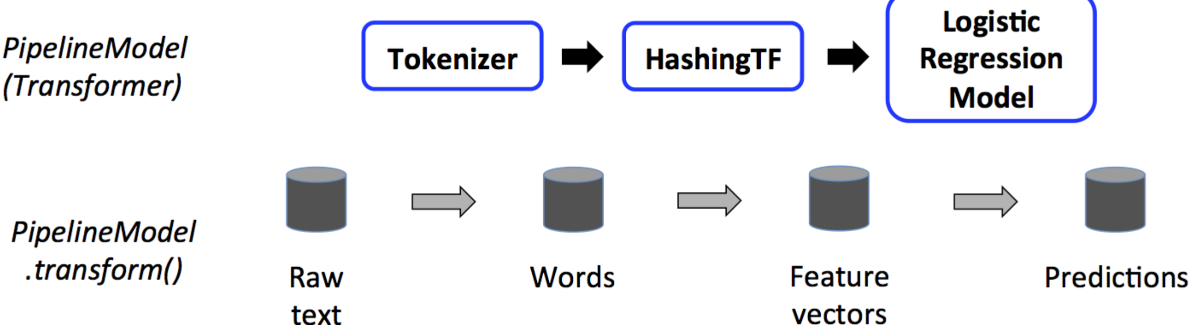


> https://spark.apache.org/docs/latest/img/ml-Pipeline.png

A fenti képen a felső sor egy három fázisú `Pipeline`-t ábrázol. Az első kettő (`Tokenizer` és `HashingTF`) `Transformer` (kékkel jelölve), míg a harmadik (`LogisticRegression`) egy `Estimator` (piros). Az alsó sor a csővezetéken átfolyó adat áramlását reprezentálja, ahol minden henger egy `DataFrame`-et jelöl. A `Pipeline.fit()` metódus az eredeti `DataFrame`-en (Raw text) kerül meghívásra, amely nyers szöveges dokumentumokat és címkéket tartalmaz. A `Tokenizer.transform()` metódus szétbontja a nyers szöveges dokumentumokat szavakra, és egy a szavakat tartalmazó új oszlopot hozzáad a `DataFrame`-hez (Words). A `HashingTF.transform()` metódus konvertálja a szavak oszlopot feature vektorokra, amely vektorokat egy új oszlopként hozzáad a `DataFrame`-hez (Feature cectors). Ezután, mivel a `LogisticRegression` egy `Estimator`, a `Pipeline` először meghívja a `LogisticRegression.fit()` metódust, hogy előállítson egy `LogisticRegressionModel`-t. Ha a `Pipeline`-nak lenne több fázisa, akkor meghívná az előállt `LogisticRegressionModel.transform()` metódust is a `DataFrame`-en mielőtt azt továbbadná a következő fázisnak.

Maga a `Pipeline` is egy `Estimator`, így miután a `Pipeline.fit()` meghívódik, előáll egy `PipelineModel`, ami egy `Transformer`. A `PipelineModel`-t aztén tesztelési időben tudjuk használni.

A `PipelineModel` tesztelés idejű felhasználása.



> https://spark.apache.org/docs/latest/img/ml-PipelineModel.png

A fenti képen látható `PipelineModel`-nek ugyanannyi fázisa van, mint az eredeti `Pipeline`-nak, azzal a különbséggel, hogy a `Pipeline` összes `Estimator` fázisa `Transformer` lett. Amikor a `PipelineModel.transform()` meghívódik a test adathalmazon, a `DataFrame` végighalad az illesztett `Pipeline` minden egyes fázisán sorban. Minden fázis `transform()` metódusa módosítja a `DataFrame`-et, majd továbbadja a következő fázisnak. A `Pipeline` és `PipelineModel` biztosítja, hogy a tanuló és teszt adatok ugyanazokon a feature feldolgozó fázisokon haladnak végig. További részletek a hivatalos dokumentációban [8] olvashatók.

A fenti ML pipeline Java nyelvű megvalósítása az MLlib segítségével a következőképp néz ki (a segéd osztályok kódját lásd a Spark GitHub repository-ban [9]):

```
/*
```

```java
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.spark.examples.ml;

// $example on$
import java.util.Arrays;

import org.apache.spark.ml.Pipeline;
import org.apache.spark.ml.PipelineModel;
import org.apache.spark.ml.PipelineStage;
import org.apache.spark.ml.classification.LogisticRegression;
import org.apache.spark.ml.feature.HashingTF;
import org.apache.spark.ml.feature.Tokenizer;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
// $example off$
import org.apache.spark.sql.SparkSession;

/**
 * Java example for simple text document 'Pipeline'.
 */
public class JavaPipelineExample {
  public static void main(String[] args) {
    SparkSession spark = SparkSession
      .builder()
      .appName("JavaPipelineExample")
      .getOrCreate();

    // $example on$
    // Prepare training documents, which are labeled.
    Dataset<Row> training = spark.createDataFrame(Arrays.asList(
      new JavaLabeledDocument(0L, "a b c d e spark", 1.0),
      new JavaLabeledDocument(1L, "b d", 0.0),
      new JavaLabeledDocument(2L, "spark f g h", 1.0),
      new JavaLabeledDocument(3L, "hadoop mapreduce", 0.0)
    ), JavaLabeledDocument.class);

    // Configure an ML pipeline, which consists of three stages: tokenizer,
hashingTF, and lr.
    Tokenizer tokenizer = new Tokenizer()
      .setInputCol("text")
      .setOutputCol("words");
    HashingTF hashingTF = new HashingTF()
      .setNumFeatures(1000)
```

```
      .setInputCol(tokenizer.getOutputCol())
      .setOutputCol("features");
    LogisticRegression lr = new LogisticRegression()
      .setMaxIter(10)
      .setRegParam(0.001);
    Pipeline pipeline = new Pipeline()
      .setStages(new PipelineStage[] {tokenizer, hashingTF, lr});

    // Fit the pipeline to training documents.
    PipelineModel model = pipeline.fit(training);

    // Prepare test documents, which are unlabeled.
    Dataset<Row> test = spark.createDataFrame(Arrays.asList(
      new JavaDocument(4L, "spark i j k"),
      new JavaDocument(5L, "l m n"),
      new JavaDocument(6L, "spark hadoop spark"),
      new JavaDocument(7L, "apache hadoop")
    ), JavaDocument.class);

    // Make predictions on test documents.
    Dataset<Row> predictions = model.transform(test);
    for (Row r : predictions.select("id", "text", "probability",
"prediction").collectAsList()) {
      System.out.println("(" + r.get(0) + ", " + r.get(1) + ") --> prob=" +
r.get(2)
        + ", prediction=" + r.get(3));
    }
    // $example off$

    spark.stop();
  }
}
```

Ellenőrző kérdések

1. Mire szolgál a Spark SQL?
2. Milyen adatforrásokból tud dolgozni a Spark SQL?
3. Milyen kapcsolatban áll egymással a Spark SQL és az Apache Hive?
4. Mik a Spark SQL megvalósítás főbb rétegei/programozói interfészei? Röviden ismertesd milyen feladatot látnak el!
5. Milyen optimalizációt végez a Catalyst? Ismersz-e más optimalizációkat, amiket a Spark alkalmaz?
6. Milyen főbb csoportokba sorolhatjuk a gépi tanuló algoritmusokat?
7. Mi a különbség a train, dev és test adathalmazok között?
8. Röviden magyarázd el az MLlib által alkalmazott `Pipeline` absztrakció lényegét!
9. Milyen típusú fázisokból állhat egy `Pipeline`? Hogyan működnek ezek a fázisok?
10. Mi a különbség a `Pipeline` és a `PipelineModel` között?

Referenciák

[1] https://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes

[2] http://hive.apache.org/

[3] https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html

[4] https://databricks.com/glossary/catalyst-optimizer

[5] https://spark.apache.org/docs/2.1.0/sql-programming-guide.html

[6] https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234

[7] https://spark.apache.org/docs/latest/ml-guide.html

[8] https://spark.apache.org/docs/latest/ml-pipeline.html

[9] https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples/ml