A pretty hot topic lately is machine learning – the inter-sectional discipline closely related to computational statistics that let's computers learn without being explicitly programmed.

It has found to be of significant use in the field of data analytics – from estimating loan and insurance risk to trying to autonomously steer a car in real-life conditions.

In the following post I would like to introduce to the reader `MLlib` – the machine learning library that is part of the Spark Framework.

One important thing about the following text – the aim is to introduce the library, not the concept and theory behind machine learning or statistics in general so an at least basic understanding of these topics is expected from the reader. Also an at least basic knowledge of Spark in general is required.

This will be based on Apache Spark 2.x API which employs the new DataFrame API as an alternative to the older `RDD` one. One of the

INDEX

TAGS

Akka (7) Akka Http (3)
Akka Persistance (3)
Akka Streams (2)
Angular2 (1) Cats (2)
Clojure (3) ClojureScript (2)
CQRS (3) DDD (3) DSL (2)
Encog (1) Event Sourcing (3)
Free Monad (2) Frontend (11)
Functional programming (6)
Futures (2) Javascript (9)
Keycloak (2) Lagom (1)

main benefits of the `DataFrame` approach is that it's easier to use and ly than the `RDD` one. Still, the `RDD` API is still present but put into maintenance mode (it will no longer be extended and will be deprecated when the `DataFrame` API will reach feature parity with it).

Introduction to MLlib

MLlib (short for Machine Learning Library) is Apache Spark's machine learning library and provides us with Spark's superb scalability and ease-of-use when trying to solve machine learning problems. Under the hood MLlib uses Breezefor it's linear algebra needs.

The library contains of a pretty extensive set of features that I will now briefly present. A more in-depth description of each feature set will be presented in the later sections.

Capabilities

Algorithms

- Regression
    - Linear
    - Generalized Linear

Machine Learning (2)
Macro (3) Microservices (1)
Mobile (2) Monix (2)
Performance (2)
RabbitMQ (2) React (8)
Reactive Platform (4)
Reactive Rabbit (3)
Reactive streams (3)
Redux (3) Scala (48)
Scala.js (4) Shapeless (2)
Slack (2) Slick (6)
Spark (3) Specs2 (1)
SQL (3) Testing (1)
Type-level programming (2)
Type class (1)
Websockets (1) Working (4)

Follow Subscribe

- Decision Tree
- Random Forest
- Gradient-boosted Tree
- Survival
- Isotonic

- Classification
    - Logistic (Binomial and Multinomial)
    - Decision Tree
    - Random Forest
    - Gradient-boosted tree
    - Multilayer Perceptron
    - Linear support vector machine
    - One-vs-All
    - Naive Bayes
- Clustering
    - K-means

    - Latent Dirichlet allocation
    - Bisecting k-means
    - Gaussian Mixture Model
- Collaborative Filtering

Featurization

- Feature extraction
- Transformation
- Dimensionality reduction
- Selection

Pipelines

- Composing Pipelines
- Constructing, evaluating and tuning machine learning Pipelines

Persistence

- Saving algorithms, models and pipelines to persistent data storage for later use
- Loading algorithms, models and pipelines from persistent data storage

The DataFrame

As mentioned before, the `DataFrame` is the new API employed in Spark versions 2.x that is supposed to be a replacement to the older `RDD` API. A `DataFrame` is a Spark `Dataset` (in short – a distributed, strongly-typed collection of data, the interface was introduced in Spark 1.6) organized into named columns (which represent the variables).

The concept is effectively the same as a table in a relational database or a data frame in R/Python, but with a set of implicit optimizations.

Characteristics

What are the main selling points and benefits of using the `DataFrame` API over the older `RDD` one? Here's a few:

- Familiarity – as mentioned beforehand, the concept is analogous [illegible]wn and used approaches of manipulating data as tables in relational databases or the data frame construct in e.g. R.
- Uniform API – the API is consistent among the languages thus we don't waste time on accommodating the differences and can focus on what's important.
- Spark SQL – it enables us accessing and manipulating the data via SQL queries and a SQL-like domain-specific language.
- Optimizations – there is a set of optimizations implemented under the hood of `Dataset` that give us more performance when handling the data.
- Multitude of possible sources – we can construct a `DataSet` from external databases, existing `RDD` s, CSV files, JSON and a multitude of other structured data sources.

Creating a DataFrame

As mentioned above – we have multiple possible sources from which we can create a `DataFrame` . To load a streaming `Dataset` from an external source we will use the `DataStreamReader` interface.

In the examples below we assume a variable named `spark` exists `ssion`. The `DataStreamReader` for the session can be obtained by calling the `read` method upon the instance.

We can add input options for the underlying data source by calling the `option` method upon the reader instance. It takes a `key` and a `value` as the argument (or a whole `Map`).

There are two approaches to loading the data: * Format-specific methods like `csv`, `jdbc`, etc. * Specifying the format explicitly with the `format` method and then calling the generic `load` method. If no format is specified `Parquet` is the default one.

Here are the most common use cases when it comes to creating a `DataFrame` and the method used:

Parquet

`Parquet` is a columnar storage format developed by Apache for projects in the `Hadoop`/`Spark` ecosystems.

We load it by calling the `load` or `parquet` methods with the path to the `Parquet` file as the argument, e.g.:


```
spark.read.load("some/path/to/file.parquet")
```

CSV

omma-separated values file. Spark can automatically infer the schema of a `CSV` file loaded.

We load it by calling the `csv` method with the path to the `CSV` file as the argument, e.g.:

```
spark.read.csv("some/path/to/file.csv")
```

JSON

The JavaScript Object Notation format most widely utilized by Web applications for asynchronous frontend/backend communication. Spark can automatically infer the schema of a `JSON` file loaded.

We load it by calling the `json` method with the path to the `JSON` file as the argument, e.g.:

```
spark.read.json("some/path/to/file.json")
```

Hive

Apache `Hive` is a data warehouse software package. For interfacing `DataFrame` s with `Hive` we need a `SparkSession` with enabled `Hive` support and all the needed dependencies in the classpath for Spark to load them automatically

We will not cover interfacing with a `Hive` data storage as this would

standing of what `Hive` is and how it works in more depth. For more information about the topic please consult the official documentation on the subject.

Database

We can easily interface with any kind of database using `JDBC`. For it to be possible You need to have the required `JDBC` driver for the database you want to interface with included in Your classpath.

We will use the `load` method mentioned before but we need to change the format from the default one (`Parquet`) to `jdbc` using the `format` method upon the reader. We can also use the `jdbc` method and passing to it a `Properties` class instance that will hold the connection properties.

We specify the `JDBC` connection properties via the `option` method mentioned before. An full list of possible options that can be passed and their descriptions are available here.

Here is an quick example how creating a `DataFrame` from a `JDBC` source could look like (example from the official

documentation):

```
val jdbcDF = spark.read
jdbc")
url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()
```

Or using the `jdbc` method:

```
val connectionProperties = new Properties()
connectionProperties.put("user", "username")
connectionProperties.put("password", "password")
val jdbcDF2 = spark.read
  .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
connectionProperties)
```

RDD

We can automatically convert a `RDD` into a `DataFrame`. The names of the arguments of the case classes will become the column names. It supports nesting complex types like `Seq` or `Array`.

All we need to do is simply call the `toDF` method on the `RDD`, i.e.:

```
val dataFrame = someRDD.toDF()
```

Defining the Schema

The data can be often inferred automatically but if for our data that option isn't available or we simply want to define it manually we have three main ways of doing so:

Casting

Explicit casting of columns from one type onto another. E.g.:

```
val dataFrame = otherDataFrame
  .withColumn("numericalColumn", dataFrame("numericalColumn").cast(DoubleType))
```

StructType

Using the `StructType` and `StructField` types to explicitly define what `DataType` is each column. E.g.:

```
val schemaStruct =
  StructType(
    StructField("intColumn", IntegerType, true) ::
    StructField("longColumn", LongType, true) ::
    StructField("booleanColumn", BooleanType, true) :: Nil)

val df = spark.read
  .schema(schemaStruct)
  .option("header", true)
  .csv(dataPath)
```

Encoders

: of Spark SQL's serialization and deserialization framework. We can use `Encoders` to provide the schema via a `case object`.

```
case class SchemaClass(intColumn: Int, longColumn: Long, booleanColumn: Boolean)

val schemaEncoded = Encoders.product[SchemaClass].schema

val df = spark.read
  .schema(schemaEncoded)
  .option("header", true)
  .csv(dataPath)
```

Saving a DataFrame

We can save a `DataFrame` to persistent storage by using the `DataFrameWriter` interface that we can obtain from a `DataFrame` by simply calling the `write` method.

Writing the `DataFrame` is almost identical in most cases, we just call the methods mentioned before on `write` instead of `read`. E.g. writing a `DataFrame` to a `JSON` file:

```
val dataFrame = spark.read.csv("someFile.csv")

dataFrame.write.json("newFile")
```

Exploring a DataFrame

ain method for inspecting the contents and structure of a `DataFrame` (or any other `Dataset`) – `show` and `printSchema`.

The `show` method comes in five versions:

- `show()` – displays the top 20 rows in tabular form.
- `show(numRows: Int)` – show the top `numRows` in tabular form.
- `show(truncate: Boolean)` – show the top 20 rows in tabular form. If `truncate` is `true` then strings longer than 20 characters will be truncated and all cells aligned right.
- `show(numRows: Int, truncate: Boolean)` – show the top `numRows` rows in tabular form. If `truncate` is `true` then strings longer than 20 characters will be truncated and all cells aligned right.
- `show(numRows: Int, truncate: Int)` – show the top `numRows` rows in tabular form. If `truncate` is more than 0 then strings longer than `truncate` characters will be truncated and all cells aligned right.

The `printSchema()` will print out the schema in tree format to the console.

DataFrame Operations

scalac erface allows us to execute operations on the data via an SQL-based DSL or by simply running SQL queries programmatically. As mentioned before the `DataFrame` is simply a `Dataset` of `Rows` thus it is not strongly typed. This is why the operations are untyped.

The `import spark.implicits._` contains implicits that let us use a richer notation when operating on the tables.

Untyped Operation

A simple example of filtering by the value of `someColumn` and then selecting `anotherColumn` as the result to be shown:

```
val result = dataFrame.filter($"someColumn" > 0).select
("anotherColumn")

result.show()
```

The `$` operator is part of the `spark.implicits` package and let's us create a `Column` reference from a `String`.

A comprehensive list of available operations can be found in the `Dataset` API documentation here.

There is also a very comprehensive set of string manipulation and
vailable. The list of them can be found here.

Running SQL Queries

We also have the option of running a SQL query programmatically with the `sql` method that takes the string with the query string as the argument.

But to do that we need to first register the `DataFrame` as a SQL `Temporary View`. This will make the `DataFrame` table be visible from the SQL query. This can be done with the `createOrReplaceTempView` method, e.g.:

```
dataFrame.createOrReplaceTempView("dataFrameTable")
```

And now running a SQL query with the `sql` method:

```
val result = spark.sql("SELECT * FROM dataFrameTable")

result.show()
```

The temporary view is session-scoped thus will disappear when the session terminates. We can create a `Global Temporary View` that will be shared among all sessions and kept alive until the application terminates. The global temporary views are tied to database thus to access them we must use the qualified name to refer it by using the `global_temp.` prefix. An example of creating and accessing such a view:

```
dataFrame.createGlobalTempView("globalDataFrameTable")

val result = spark.sql("SELECT * FROM global_temp.globalDataFrameTable")

result.show()
```

Pipelines

The `Pipeline` concept revolves around the idea of providing a uniform API to create and compose together machine learning data-transformation pipelines to create a single, concise workflow. It also provides us with the option to persist them and use an already existing one that we created and saved earlier. The concept is analogous to stream-processing in e.g. `Akka Streams`.

A `Pipeline` can consist of the following elements:

- Transformer – an abstraction of `DataFrame` transformers. Consists of a `transform` function that maps a `DataFrame` into new one by

e.g. adding a column, changing the rows of a specific column,
label based on the feature vector.

- Estimator – an abstraction of algorithms that fit or train or data (e.g. regression algorithms). Consists of a `fit` function that maps a `DataFrame` into a `Model`.

Additionally `Transformer` and `Estimator` share a common API for specifying their parameters – `Parameter` as an alternative to using setters. More information about the `Parameter` concept can be found here

Pipeline

A `Pipeline` in essence is an ordered array of stages. As mentioned before, a stage is either a `Transformer` or an `Estimator`. Of course we can easily tell from looking at the domain and co-domain of both that a `Pipeline` can consist of many `Transform` stages but only one `Estimator` stage that must be at the end of the `Pipeline`.

An example `Pipeline` for some simple regression task: 1. Converting categorical features into indexes. 2. Normalizing the vectors in the frame. 3. Linear regression.

Saving/Loading

scalac ave a created `Pipeline` or `Model` for later use. Not all `Transform` and `Estimator` types are supported so checking their docs for specific information about it is a good idea. Most of the basic transformers and models are supported. The methods:

- `save(path: String)` – save the `Model` / `Pipeline` to the location pointed by `path`
- `load(path: String)` – load a `Model` / `Pipeline` from the location pointed by `path`

Example

Here is a short example of how to create a `Pipeline` (note that the `setStages` method takes an `Array` as the argument):

```
val indexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexedFeatures")
  .setMaxCategories(5)

val normalizer = new Normalizer()
  .setInputCol("features")
  .setOutputCol("normalizedFeatures")
  .setP(1.0)

val lr = new LinearRegression()
  .setMaxIter(100)
```

```
  .setRegParam(0.5)
  .setElasticNetParam(0.5)

val pipeline = new Pipeline()
  .setStages(Array(indexer, normalizer, lr))
```

Transformers and Estimators in Spark

MLlib comes with an extensive set of `Transformer` and algorithm `Estimator` elements that we can use in our machine learning workflows. The documentation provided for each of them is really excellent and I suggest checking it out. You can find it under the following links:

- Extracting, transforming and selecting features
- Classification and regression

The regression/classification algorithms in the library operate on two `Double` -value vectors – the feature vector and the label vector. Thus for categorical values we need to transform the columns using an indexer and the multiple feature column values need to be collected into a single vector (e.g. by using a VectorAssembler).

Spark also offers us a way to define our own `Transformer` and `Estimator` components if the ones provided

aren't enough. For further information I would suggest
ng the Pipeline by Tomasz Sosiński.

Example

Finally I would like to present an example of a full-fledged code for doing regression on a real-world dataset (albeit we'll be only looking at a small portion of it).

We'll try to tackle a regression problem of predicting the price of a wine based on two variables – it's WineEnthusiast rating and the country where it was made. We'll use this data set for doing so. The unpacked file is renamed to `wine-data.csv` and moved to the application's working directory.

The `WineEnthusiast` variable is closer in definition to an ordinal variable if you look at it's values and variable description but we'll treat it as a 'Double' for the sake of the example. Country is a categorical (nominal) value thus need to be indexed for the feature vector. Then we'll collect the new columns into a single vector named `features` using the mentioned before `VectorAssembler`.


```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
```

```
import org.apache.spark.ml.feature.{StringIndexer, VectorAssembler}
          apache.spark.ml.regression.GBTRegressor
          apache.spark.sql.types.{DoubleType, StringType,
          , StructType}
import org.apache.spark.sql.{Encoders, SparkSession}

object Main {

    def main(args: Array[String]) = {

        val spark = SparkSession.builder
            .appName("Wine Price Regression")
            .master("local")
            .getOrCreate()

        //We'll define a partial schema with the values we are
interested in. For the sake of the example points is a Double
        val schemaStruct = StructType(
            StructField("points", DoubleType) ::
            StructField("country", StringType) ::
            StructField("price", DoubleType) :: Nil
        )

        //We read the data from the file taking into account there's a
header.
        //na.drop() will return rows where all values are non-null.
        val df = spark.read
            .option("header", true)
            .schema(schemaStruct)
            .csv("wine-data.csv")
            .na.drop()

        //We'll split the set into training and test data
        val Array(trainingData, testData) = df.randomSplit(Array(0.8,
0.2))

        val labelColumn = "price"
```

```scala

        //We define two StringIndexers for the categorical variables

        val countryIndexer = new StringIndexer()
            .setInputCol("country")
            .setOutputCol("countryIndex")

        //We define the assembler to collect the columns into a new column with a single vector - "features"
        val assembler = new VectorAssembler()
            .setInputCols(Array("points", "countryIndex"))
            .setOutputCol("features")

        //For the regression we'll use the Gradient-boosted tree estimator
        val gbt = new GBTRegressor()
            .setLabelCol(labelColumn)
            .setFeaturesCol("features")
            .setPredictionCol("Predicted " + labelColumn)
            .setMaxIter(50)

        //We define the Array with the stages of the pipeline
        val stages = Array(
            countryIndexer,
            assembler,
            gbt
        )

        //Construct the pipeline
        val pipeline = new Pipeline().setStages(stages)

        //We fit our DataFrame into the pipeline to generate a model
        val model = pipeline.fit(trainingData)

        //We'll make predictions using the model and the test data
        val predictions = model.transform(testData)

```

```
        //This will evaluate the error/deviation of the regression
using the Root Mean Squared deviation
        evaluator = new RegressionEvaluator()
            .setLabelCol(labelColumn)
            .setPredictionCol("Predicted " + labelColumn)
            .setMetricName("rmse")

        //We compute the error using the evaluator
        val error = evaluator.evaluate(predictions)

        println(error)

        spark.stop()
    }
}
```

Afterword

I hope that the article was helpful in understanding the basics behind MLlib and how to utilize it in Your machine learning endeavours. As we could see the library (and Spark in general) provide us with a well designed API and workflow for doing machine learning. Of course this text was meant as an introduction thus doesn't exhaust the subject. But, as mentioned before, Spark provides us with great documentation that let's us pursue it in more depth.

In the last section I've provided some links that I think should prove to be very useful for expanding our knowledge further on the subject.

Happy coding ;)

Cheers, Marcin

Useful Links

- Official MLlib Guide
- Spark SQL, DataFrames and Datasets Guide
- Dataset API
- SQL functions available
- Feature Transformers Documentation
- Classification and Regression Algorithms Documentation
- Extending the Pipeline

Do you like this post? Want to stay updated? Follow us on Twitter or subscribe to our Feed.

Author profile

Marcin Gorczyński

Visit website

main non-IT hobbies and interests include music, playing guitar(s) and piano, politics, philosophy and ancient history (mainly Greek).