Data Science in Spark

mit sparklyr

Schummelzettel

RStudio

Data Science Arbeitsablauf mit Spark + sparklyr

Importieren
- R-DataFrame exportieren
- Datei einlesen
- Vorhandene Hive-Tabelle einlesen

Aufräumen
- dplyr Verb
- Direct Spark SQL (DBI)
- SDF Funktion (Scala API)

Daten bändigen

Verstehen

Umformen
Umform-Funktion

Visualisieren
Daten fürs Zeichen in R einbinden

Modellieren
- Spark MLlib
- H2O Erweiterung

Mitteilen
- Daten in R einsammeln
- Zeichnungen, Dokumente und Apps teilen

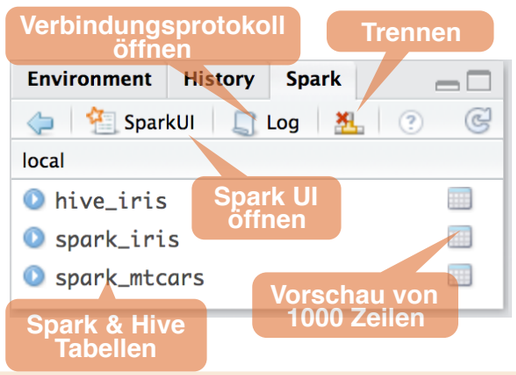
R for Data Science, Grolemund & Wickham

Einführung

sparklyr ist eine R-Schnittstelle für Apache Spark™. Sie stellt ein komplettes **dplyr**-Backend zur Verfügung und bietet die Möglichkeit an, direkt mit **Spark SQL** abzufragen. Mit sparklyr kann das verteilte maschinelle Lernen entweder mit **Spark MLlib** oder **H2O** Sparkling Water in Szene gesetzt werden.

Ab **Version 1.044** ist **die Unterstützung für das sparklyr-Paket in RStudio Desktop, Server und Pro integriert**. Direkt aus der IDE heraus können Verbindungen zu Spark Clustern und lokalen Spark-Instanzen hergestellt und verwaltet werden.

Integration von sparklyr in RStudio



Erste Schritte

Lokal-Modus

Einfache Einrichtung. Kein Cluster notwendig

1. Spark lokal installieren:
   spark_install ("2.0.1")
2. Verbindung herstellen:
   sc <- spark_connect (master = "local")

Auf Mesos Managed Cluster

1. RStudio Server oder Pro auf einem vorhandenen Knoten installieren
2. Spark-Cluster-Pfad festlegen
3. Verbindung herstellen:
   spark_connect(master="[Mesos-URL]", version = "1.6.2", spark_home = [Spark-Cluster-Pfad])

Unter Livy (experimentell)

1. Livy REST-Anwendung sollte bereits auf dem Cluster laufen
2. Mit dem Cluster verbinden:
   sc <- spark_connect(master = "http://host:port" , method = "livy")

Auf YARN Managed Cluster

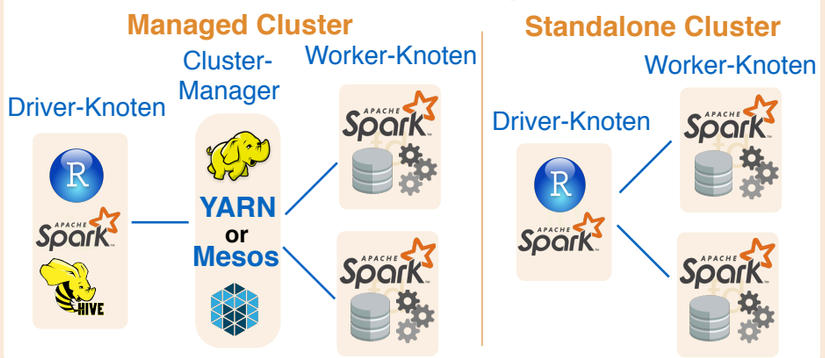
1. RStudio Server oder Pro auf einem vorhandenen Knoten (vorzugsweise Eckknoten) installieren
2. Spark-Cluster-Pfad festlegen (gewöhnlich unter "/usr/lib/spark")
3. Verbindung herstellen:
   spark_connect(master="yarn-client", version = "1.6.2", spark_home = [Spark-Cluster-Pfad])

Auf Spark Standalone Cluster

1. RStudio Server oder Pro auf einem vorhandenen Knoten oder auf einem Server in demselben LAN installieren
2. Spark lokal installieren:
   spark_install (version = "2.0.1")
3. Verbindung herstellen:
   spark_connect(master="spark://host:port", version = "2.0.1", spark_home = spark_home_dir())

Cluster-Installation

Cluster-Installation-Optionen



Tuning von Spark

Beispiel-Konfiguration

```
config <- spark_config()
config$spark.executor.cores <- 2
config$spark.executor.memory <- "4G"
sc <- spark_connect (master = "yarn-client",
config = config, version = "2.0.1")
```

Wichtige Tuning-Parameter
mit Default-Werten

- spark.yarn.am.cores
- spark.yarn.am.memory *512m*

Wichtige Tuning-Parameter
mit Default-Werten *fortgesetzt*

- spark.executor.heartbeatInterval *10s*
- spark.network.timeout *120s*
- spark.executor.memory *1g*
- spark.executor.cores *1*
- spark.executor.extraJavaOptions
- spark.executor.instances
- sparklyr.shell.executor-memory
- sparklyr.shell.driver-memory

sparklyr im Einsatz

Ein kurzes Beispiel für eine Datenanlyse mit Apache Spark, R und sparklyr im lokalen Modus

```
library(sparklyr); library(dplyr); library(ggplot2);
library(tidyr);
set.seed(100)
```
Spark lokal installieren

```
spark_install("2.0.1")
```
Lokale Verbindung herstellen

```
sc <- spark_connect(master = "local")

import_iris <- copy_to(sc, iris, "spark_iris",
  overwrite = TRUE)
```
Daten in den Spark-Speicher kopieren

```
partition_iris <- sdf_partition(
  import_iris, training=0.5,
  testing=0.5)
```
Spark-Daten partitionieren

```
sdf_register(partition_iris,
c("spark_iris_training","spark_iris_test"))
```
Pro Partition einen Tabellennamen zuweisen

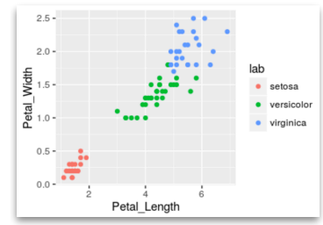
```
tidy_iris <- tbl(sc,"spark_iris_training") %>%
  select(Species, Petal_Length, Petal_Width)

model_iris <- tidy_iris %>%
  ml_decision_tree(response="Species",
  features=c("Petal_Length","Petal_Width"))
```
Spark ML Entscheidungs-baum-Modell

```
test_iris <- tbl(sc,"spark_iris_test")
```
Referenz auf Spark-Tabelle erstellen

```
pred_iris <- sdf_predict(
  model_iris,  test_iris) %>%
  collect
```
Daten für die Visualisierung in R vorbereiten

```
pred_iris %>%
  inner_join(data.frame(prediction=0:2,
  lab=model_iris$model.parameters$labels)) %>%
  ggplot(aes(Petal_Length, Petal_Width, col=lab)) +
  geom_point()
```




```
spark_disconnect(sc)
```
Verbindung trennen

Übersetzt von Ke Zhang • ke@2cn.de

Weitere Informationen auf spark.rstudio.com • Paketversion 0.5 • Aktualisiert am: 21.12.16

Importieren

DataFrame nach Spark kopieren

sdf_copy_to(sc, iris, "spark_iris")

sdf_copy_to(sc, x, name, memory, repartition, overwrite**)**

Spark SQL Befehle

DBI::**dbWriteTable**(
sc, "spark_iris", iris)

DBI::**dbWriteTable(**conn, name, value**)**

Eine Datei in Spark einlesen

Argumente für alle Funktionen:
sc, name, path, options = list(), repartition = 0, memory = TRUE, overwrite = TRUE

CSV **spark_read_csv(** header = TRUE, columns = NULL, infer_schema = TRUE, delimiter = ",", quote = "\"", escape = "\\", charset = "UTF-8", null_value = NULL**)**

JSON **spark_read_json()**

PARQUET **spark_read_parquet()**

Von einer Hive-Tabelle

my_var <- **tbl_cache**(sc, name= "hive_iris")

tbl_cache(sc, name, force = TRUE**)**
Tabelle in den Cache laden

my_var <- **dplyr::tbl**(sc, name= "hive_iris")

dplyr::**tbl(**scr, …**)**
Referenz auf die Tabelle erstellen ohne sie direkt in den Speicher zu laden.

Daten bändigen

Spark SQL über dplyr Verben

Übersetzt als Spark SQL-Anweisungen

```
my_table <- my_var %>%
  filter(Species=="setosa") %>%
  sample_n(10)
```

Direct Spark SQL Befehle

my_table <- DBI::**dbGetQuery**(sc , "SELECT * FROM iris LIMIT 10")

DBI::**dbGetQuery(**conn, statement**)**

Scala API über SDF-Funktionen

sdf_mutate(.data**)**
Funktioniert ähnlich wie die dplyr-mutate-Methode

sdf_partition(x, ..., weights = NULL, seed = sample (.Machine$integer.max, 1**))**

sdf_partition*(x, training = 0.5, test = 0.5)*

sdf_register(x, name = NULL**)**
Spark DataFrame einen Namen zuweisen

sdf_sample(x, fraction = 1, replacement = TRUE, seed = NULL**)**

sdf_sort(x, columns**)**
Eine oder mehrere Spalten in aufsteigender Reihenfolge sortieren

sdf_with_unique_id(x, id = "id"**)**
Eindeutige ID-Spalte hinzufügen

sdf_predict(object, newdata**)**
Spark DataFrame mit prognostizierten Werten

ML Transformatoren

ft_binarizer(my_table,input.col="Petal_Length", output.col="petal_large", threshold=1.2)

Argumente für alle Funktionen:
x, input.col = NULL, output.col = NULL

ft_binarizer(threshold = 0.5**)**
Umwandlung in Dual-Wert basierend auf Schwellenwert

ft_bucketizer(splits**)**
Numerische in diskretisierte Spalte

ft_discrete_cosine_transform(inverse = FALSE**)**
Zeitbereich in Frequenzbereich

ft_elementwise_product(scaling.col**)**
Elementweises Produkt von 2 Spalten

ft_index_to_string()
Index-Spalte in String-Spalte

ft_one_hot_encoder()
Kontinuierliche in binäre Vektoren

ft_quantile_discretizer(n.buckets = 5L**)**
Kontinuierliche zu in Bins unterteilten kategorischen Werten

ft_sql_transformer(sql**)**

ft_string_indexer(params = NULL**)**
String-Spalte in Index-Spalte

ft_vector_assembler()
Vektoren zu einem Einzelzeilen-Vektor vereinigen

Visualisieren & Mitteilen

Daten in den R-Speicher laden

```
r_table <- collect(my_table)
plot(Petal_Width~Petal_Length, data=r_table)
```

dplyr::**collect(**x**)**
Spark DataFrame in R-DataFrame laden

sdf_read_column(x, column**)**
Inhalt einer einzelnen R-Spalte auslesen

Von Spark auf Dateisystem speichern

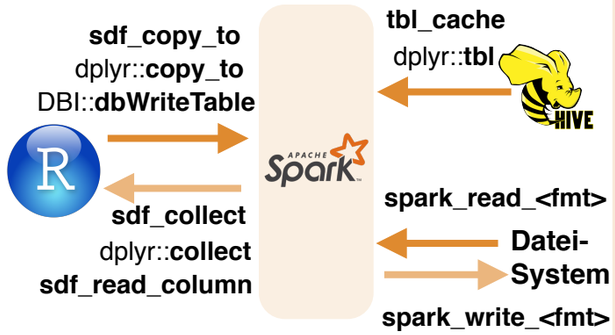
Argumente für alle Funktionen: x, path

CSV **spark_read_csv(** header = TRUE, delimiter = ",", quote = "\"", escape = "\\", charset = "UTF-8", null_value = NULL**)**

JSON **spark_read_json(**mode = NULL**)**

PARQUET **spark_read_parquet(**mode = NULL**)**

Lesen & Schreiben in Apache Spark



Erweiterungen

Erstellen eines R-Pakets, welches die volle Spark-API anbietet & Schnittstellen zu Spark-Paketen bereitstellt

Kernklassen

spark_connection() Referenz auf Verbindung zwischen R und dem Spark-Shell-Prozess

spark_jobj() Instanz eines Remote-Spark-Objekts

spark_dataframe() Instanz eines Remote-Spark-DataFrame-Objekts

Spark-Aufruf in R

invoke() Methode eines Java-Objekts aufrufen

invoke_new() Ein neues Java-Objekt über Konstruktor-Aufruf erstellen

invoke_static() Eine statische Methode aufrufen

Erweiterungen für das maschinelle Lernen

ml_create_dummy_variables()
ml_prepare_dataframe()
ml_prepare_response_features_intercept()
ml_options()
ml_model()

Modellieren (MLlib)

ml_decision_tree(my_table , response="Species", features= c("Petal_Length" , "Petal_Width"))

ml_als_factorization(x, rating.column = "rating", user.column = "user", item.column = "item", rank = 10L, regularization.parameter = 0.1, iter.max = 10L, ml.options = ml_options()**)**

ml_decision_tree(x, response, features, max.bins = 32L, max.depth = 5L, type = c("auto", "regression", "classification"), ml.options = ml_options()**)**
Gleiche Optionen für: **ml_gradient_boosted_trees**

ml_generalized_linear_regression(x, response, features, intercept = TRUE, family = gaussian(link = "identity"), iter.max = 100L, ml.options = ml_options()**)**

ml_kmeans(x, centers, iter.max = 100, features = dplyr::tbl_vars(x), compute.cost = TRUE, tolerance = 1e-04, ml.options = ml_options()**)**

ml_lda(x, features = dplyr::tbl_vars(x), k = length(features), alpha = (50/k) + 1, beta = 0.1 + 1, ml.options = ml_options()**)**

ml_linear_regression(x, response, features, intercept = TRUE, alpha = 0, lambda = 0, iter.max = 100L, ml.options = ml_options()**)**
Gleiche Optionen für: **ml_logistic_regression**

ml_multilayer_perceptron(x, response, features, layers, iter.max = 100, seed = sample(.Machine$integer.max, 1), ml.options = ml_options()**)**

ml_naive_bayes(x, response, features, lambda = 0, ml.options = ml_options()**)**

ml_one_vs_rest(x, classifier, response, features, ml.options = ml_options()**)**

ml_pca(x, features = dplyr::tbl_vars(x), ml.options = ml_options()**)**

ml_random_forest(x, response, features, max.bins = 32L, max.depth = 5L, num.trees = 20L, type = c("auto", "regression", "classification"), ml.options = ml_options()**)**

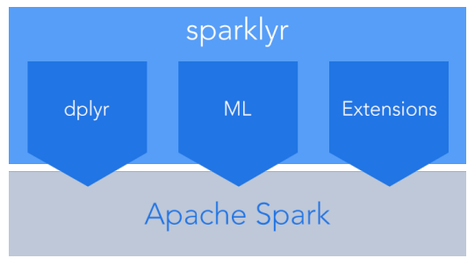
ml_survival_regression(x, response, features, intercept = TRUE,censor = "censor", iter.max = 100L, ml.options = ml_options()**)**

ml_binary_classification_eval(predicted_tbl_spark, label, score, metric = "areaUnderROC"**)**

ml_classification_eval(predicted_tbl_spark, label, predicted_lbl, metric = "f1"**)**

ml_tree_feature_importance(sc, model**)**

sparklyr ist eine R-Schnittstelle für Apache Spark



Weitere Informationen auf spark.rstudio.com • Paketversion 0.5 • Aktualisiert am: 21.12.16
Übersetzt von Ke Zhang • ke@2cn.de