1. Perform the below given activities:

a. Take a sample data set of your choice

> sample(1:6, 10, replace=TRUE)

[1] 2 2 5 3 5 3 5 6 3 5

> set.seed(1)

> sample(1:6, 10, replace=TRUE)

[1] 2 3 4 6 2 6 6 4 4 1

> sample(1:6, 10, replace=TRUE)

[1] 2 2 5 3 5 3 5 6 3 5

> set.seed(1)

> sample(1:6, 10, replace=TRUE)

[1] 2 3 4 6 2 6 6 4 4 1

> set.seed(123)

> index <- sample(1:nrow(iris), 5)

> index

[1] 44 119 62 133 142

> iris[index, ]

Sepal.Length Sepal.Width Petal.Length Petal.Width Species

44 5.0 3.5 1.6 0.6 setosa

119 7.7 2.6 6.9 2.3 virginica

62 5.9 3.0 4.2 1.5 versicolor

133 6.4 2.8 5.6 2.2 virginica

142 6.9 3.1 5.1 2.3 virginica

b. Apply random forest, logistic regression using Spark R

**Logistic regression**

Logistic regression is a popular method to predict a categorical response. It is a special case of [Generalized Linear models](https://en.wikipedia.org/wiki/Generalized_linear_model) that predicts the probability of the outcomes. In spark.ml logistic regression can be used to predict a binary outcome by using binomial logistic regression, or it can be used to predict a multiclass outcome by using multinomial logistic regression. Use the family parameter to select between these two algorithms, or leave it unset and Spark will infer the correct variant.

Multinomial logistic regression can be used for binary classification by setting the family param to “multinomial”. It will produce two sets of coefficients and two intercepts.

When fitting LogisticRegressionModel without intercept on dataset with constant nonzero column, Spark MLlib outputs zero coefficients for constant nonzero columns. This behavior is the same as R glmnet but different from LIBSVM.

### Binomial logistic regression

For more background and more details about the implementation of binomial logistic regression, refer to the documentation of [logistic regression in spark.mllib](https://spark.apache.org/docs/2.2.0/mllib-linear-methods.html#logistic-regression).

**Examples**

The following example shows how to train binomial and multinomial logistic regression models for binary classification with elastic net regularization. elasticNetParam corresponds to αα and regParam corresponds to λλ.

*# Load training data*

df <- read.df("data/mllib/sample\_libsvm\_data.txt", **source** = "libsvm")

training <- df

test <- df

*# Fit an binomial logistic regression model with spark.logit*

model <- spark.logit(training, label ~ features, maxIter = 10, regParam = 0.3, elasticNetParam = 0.8)

*# Model summary*

summary(model)

*# Prediction*

predictions <- predict(model, test)

head(predictions)

### Multinomial logistic regression

Multiclass classification is supported via multinomial logistic (softmax) regression. In multinomial logistic regression, the algorithm produces KK sets of coefficients, or a matrix of dimension K×JK×J where KK is the number of outcome classes and JJ is the number of features. If the algorithm is fit with an intercept term then a length KK vector of intercepts is available.

Multinomial coefficients are available as coefficientMatrix and intercepts are available as interceptVector.

coefficients and intercept methods on a logistic regression model trained with multinomial family are not supported. Use coefficientMatrix and interceptVector instead.

The conditional probabilities of the outcome classes k∈1,2,…,Kk∈1,2,…,K are modeled using the softmax function.

P(Y=k|X,βk,β0k)=eβk⋅X+β0k∑K−1k′=0eβk′⋅X+β0k′P(Y=k|X,βk,β0k)=eβk⋅X+β0k∑k′=0K−1eβk′⋅X+β0k′

We minimize the weighted negative log-likelihood, using a multinomial response model, with elastic-net penalty to control for overfitting.

minβ,β0−[∑i=1Lwi⋅logP(Y=yi|xi)]+λ[12(1−α)||β||22+α||β||1]minβ,β0−[∑i=1Lwi⋅log⁡P(Y=yi|xi)]+λ[12(1−α)||β||22+α||β||1]

For a detailed derivation please see [here](https://en.wikipedia.org/wiki/Multinomial_logistic_regression#As_a_log-linear_model).

**Examples**

The following example shows how to train a multiclass logistic regression model with elastic net regularization.

*# Load training data*

df <- read.df("data/mllib/sample\_multiclass\_classification\_data.txt", **source** = "libsvm")

training <- df

test <- df

*# Fit a multinomial logistic regression model with spark.logit*

model <- spark.logit(training, label ~ features, maxIter = 10, regParam = 0.3, elasticNetParam = 0.8)

*# Model summary*

summary(model)

*# Prediction*

predictions <- predict(model, test)

head(predictions)

## Decision tree classifier

Decision trees are a popular family of classification and regression methods. More information about the spark.mlimplementation can be found further in the [section on decision trees](https://spark.apache.org/docs/2.2.0/ml-classification-regression.html#decision-trees).

**Examples**

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the DataFrame which the Decision Tree algorithm can recognize.

## Random forest classifier

Random forests are a popular family of classification and regression methods. More information about the spark.mlimplementation can be found further in the [section on random forests](https://spark.apache.org/docs/2.2.0/ml-classification-regression.html#random-forests).

**Examples**

The following examples load a dataset in LibSVM format, split it into training and test sets, train on the first dataset, and then evaluate on the held-out test set. We use two feature transformers to prepare the data; these help index categories for the label and categorical features, adding metadata to the DataFrame which the tree-based algorithms can recognize.

*# Load training data*

df <- read.df("data/mllib/sample\_libsvm\_data.txt", **source** = "libsvm")

training <- df

test <- df

*# Fit a random forest classification model with spark.randomForest*

model <- spark.randomForest(training, label ~ features, "classification", numTrees = 10)

*# Model summary*

summary(model)

*# Prediction*

predictions <- predict(model, test)

head(predictions)

c. Predict for new dataset

# SparkDataFrame

A SparkDataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R, but with richer optimizations under the hood. SparkDataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing local R data frames.

All of the examples on this page use sample data included in R or the Spark distribution and can be run using the ./bin/sparkR shell.

## Starting Up: SparkSession

The entry point into SparkR is the SparkSession which connects your R program to a Spark cluster. You can create a SparkSession using sparkR.session and pass in options such as the application name, any spark packages depended on, etc. Further, you can also work with SparkDataFrames via SparkSession. If you are working from the sparkR shell, the SparkSession should already be created for you, and you would not need to call sparkR.session.

sparkR.session()

## Starting Up from RStudio

You can also start SparkR from RStudio. You can connect your R program to a Spark cluster from RStudio, R shell, Rscript or other R IDEs. To start, make sure SPARK\_HOME is set in environment (you can check [Sys.getenv](https://stat.ethz.ch/R-manual/R-devel/library/base/html/Sys.getenv.html)), load the SparkR package, and call sparkR.session as below. It will check for the Spark installation, and, if not found, it will be downloaded and cached automatically. Alternatively, you can also run install.sparkmanually.

In addition to calling sparkR.session, you could also specify certain Spark driver properties. Normally these [Application properties](https://spark.apache.org/docs/2.1.0/configuration.html#application-properties) and [Runtime Environment](https://spark.apache.org/docs/2.1.0/configuration.html#runtime-environment) cannot be set programmatically, as the driver JVM process would have been started, in this case SparkR takes care of this for you. To set them, pass them as you would other configuration properties in the sparkConfig argument to sparkR.session().

**if** (nchar(Sys.getenv("SPARK\_HOME")) < 1) {

Sys.setenv(SPARK\_HOME = "/home/spark")

}

**library**(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK\_HOME"), "R", "lib")))

sparkR.session(master = "local[\*]", sparkConfig = list(spark.driver.memory = "2g"))

## Creating SparkDataFrames

With a SparkSession, applications can create SparkDataFrames from a local R data frame, from a [Hive table](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#hive-tables), or from other [data sources](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#data-sources).

### From local data frames

The simplest way to create a data frame is to convert a local R data frame into a SparkDataFrame. Specifically we can use as.DataFrame or createDataFrame and pass in the local R data frame to create a SparkDataFrame. As an example, the following creates a SparkDataFrame based using the faithful dataset from R.

df <- as.DataFrame(faithful)

*# Displays the first part of the SparkDataFrame*

head(df)

*## eruptions waiting*

*##1 3.600 79*

*##2 1.800 54*

*##3 3.333 74*

### From Data Sources

SparkR supports operating on a variety of data sources through the SparkDataFrame interface. This section describes the general methods for loading and saving data using Data Sources. You can check the Spark SQL programming guide for more [specific options](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#manually-specifying-options) that are available for the built-in data sources.

The general method for creating SparkDataFrames from data sources is read.df. This method takes in the path for the file to load and the type of data source, and the currently active SparkSession will be used automatically. SparkR supports reading JSON, CSV and Parquet files natively, and through packages available from sources like [Third Party Projects](http://spark.apache.org/third-party-projects.html), you can find data source connectors for popular file formats like Avro. These packages can either be added by specifying --packages with spark-submit or sparkR commands, or if initializing SparkSession with sparkPackages parameter when in an interactive R shell or from RStudio.

sparkR.session(sparkPackages = "com.databricks:spark-avro\_2.11:3.0.0")

We can see how to use data sources using an example JSON input file. Note that the file that is used here is not a typical JSON file. Each line in the file must contain a separate, self-contained valid JSON object. For more information, please see [JSON Lines text format, also called newline-delimited JSON](http://jsonlines.org/). As a consequence, a regular multi-line JSON file will most often fail.

people <- read.df("./examples/src/main/resources/people.json", "json")

head(people)

*## age name*

*##1 NA Michael*

*##2 30 Andy*

*##3 19 Justin*

*# SparkR automatically infers the schema from the JSON file*

printSchema(people)

*# root*

*# |-- age: long (nullable = true)*

*# |-- name: string (nullable = true)*

*# Similarly, multiple files can be read with read.json*

people <- read.json(c("./examples/src/main/resources/people.json", "./examples/src/main/resources/people2.json"))

The data sources API natively supports CSV formatted input files. For more information please refer to SparkR [read.df](https://spark.apache.org/docs/2.1.0/api/R/read.df.html) API documentation.

df <- read.df(csvPath, "csv", header = "true", inferSchema = "true", na.strings = "NA")

The data sources API can also be used to save out SparkDataFrames into multiple file formats. For example we can save the SparkDataFrame from the previous example to a Parquet file using write.df.

write.df(people, path = "people.parquet", **source** = "parquet", mode = "overwrite")

### From Hive tables

You can also create SparkDataFrames from Hive tables. To do this we will need to create a SparkSession with Hive support which can access tables in the Hive MetaStore. Note that Spark should have been built with [Hive support](https://spark.apache.org/docs/2.1.0/building-spark.html#building-with-hive-and-jdbc-support) and more details can be found in the [SQL programming guide](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#starting-point-sparksession). In SparkR, by default it will attempt to create a SparkSession with Hive support enabled (enableHiveSupport = TRUE).

sparkR.session()

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

*# Queries can be expressed in HiveQL.*

results <- sql("FROM src SELECT key, value")

*# results is now a SparkDataFrame*

head(results)

*## key value*

*## 1 238 val\_238*

*## 2 86 val\_86*

*## 3 311 val\_311*

## SparkDataFrame Operations

SparkDataFrames support a number of functions to do structured data processing. Here we include some basic examples and a complete list can be found in the [API](https://spark.apache.org/docs/2.1.0/api/R/index.html) docs:

### Selecting rows, columns

*# Create the SparkDataFrame*

df <- as.DataFrame(faithful)

*# Get basic information about the SparkDataFrame*

df

*## SparkDataFrame[eruptions:double, waiting:double]*

*# Select only the "eruptions" column*

head(select(df, df$eruptions))

*## eruptions*

*##1 3.600*

*##2 1.800*

*##3 3.333*

*# You can also pass in column name as strings*

head(select(df, "eruptions"))

*# Filter the SparkDataFrame to only retain rows with wait times shorter than 50 mins*

head(filter(df, df$waiting < 50))

*## eruptions waiting*

*##1 1.750 47*

*##2 1.750 47*

*##3 1.867 48*

### Grouping, Aggregation

SparkR data frames support a number of commonly used functions to aggregate data after grouping. For example we can compute a histogram of the waiting time in the faithful dataset as shown below

*# We use the `n` operator to count the number of times each waiting time appears*

head(summarize(groupBy(df, df$waiting), count = n(df$waiting)))

*## waiting count*

*##1 70 4*

*##2 67 1*

*##3 69 2*

*# We can also sort the output from the aggregation to get the most common waiting times*

waiting\_counts <- summarize(groupBy(df, df$waiting), count = n(df$waiting))

head(arrange(waiting\_counts, desc(waiting\_counts$count)))

*## waiting count*

*##1 78 15*

*##2 83 14*

*##3 81 13*

### Operating on Columns

SparkR also provides a number of functions that can directly applied to columns for data processing and during aggregation. The example below shows the use of basic arithmetic functions.

*# Convert waiting time from hours to seconds.*

*# Note that we can assign this to a new column in the same SparkDataFrame*

df$waiting\_secs <- df$waiting \* 60

head(df)

*## eruptions waiting waiting\_secs*

*##1 3.600 79 4740*

*##2 1.800 54 3240*

*##3 3.333 74 4440*

### Applying User-Defined Function

In SparkR, we support several kinds of User-Defined Functions:

#### Run a given function on a large dataset using dapply or dapplyCollect

##### dapply

Apply a function to each partition of a SparkDataFrame. The function to be applied to each partition of the SparkDataFrame and should have only one parameter, to which a data.frame corresponds to each partition will be passed. The output of function should be a data.frame. Schema specifies the row format of the resulting a SparkDataFrame. It must match to [data types](https://spark.apache.org/docs/2.1.0/sparkr.html#data-type-mapping-between-r-and-spark) of returned value.

*# Convert waiting time from hours to seconds.*

*# Note that we can apply UDF to DataFrame.*

schema <- structType(structField("eruptions", "double"), structField("waiting", "double"),

structField("waiting\_secs", "double"))

df1 <- dapply(df, **function**(x) { x <- cbind(x, x$waiting \* 60) }, schema)

head(collect(df1))

*## eruptions waiting waiting\_secs*

*##1 3.600 79 4740*

*##2 1.800 54 3240*

*##3 3.333 74 4440*

*##4 2.283 62 3720*

*##5 4.533 85 5100*

*##6 2.883 55 3300*

##### dapplyCollect

Like dapply, apply a function to each partition of a SparkDataFrame and collect the result back. The output of function should be a data.frame. But, Schema is not required to be passed. Note that dapplyCollect can fail if the output of UDF run on all the partition cannot be pulled to the driver and fit in driver memory.

*# Convert waiting time from hours to seconds.*

*# Note that we can apply UDF to DataFrame and return a R's data.frame*

ldf <- dapplyCollect(

df,

**function**(x) {

x <- cbind(x, "waiting\_secs" = x$waiting \* 60)

})

head(ldf, 3)

*## eruptions waiting waiting\_secs*

*##1 3.600 79 4740*

*##2 1.800 54 3240*

*##3 3.333 74 4440*

#### Run a given function on a large dataset grouping by input column(s) and using gapply or gapplyCollect

##### gapply

Apply a function to each group of a SparkDataFrame. The function is to be applied to each group of the SparkDataFrame and should have only two parameters: grouping key and R data.frame corresponding to that key. The groups are chosen from SparkDataFrames column(s). The output of function should be a data.frame. Schema specifies the row format of the resulting SparkDataFrame. It must represent R function’s output schema on the basis of Spark [data types](https://spark.apache.org/docs/2.1.0/sparkr.html#data-type-mapping-between-r-and-spark). The column names of the returned data.frame are set by user.

*# Determine six waiting times with the largest eruption time in minutes.*

schema <- structType(structField("waiting", "double"), structField("max\_eruption", "double"))

result <- gapply(

df,

"waiting",

**function**(key, x) {

y <- data.frame(key, max(x$eruptions))

},

schema)

head(collect(arrange(result, "max\_eruption", decreasing = **TRUE**)))

*## waiting max\_eruption*

*##1 64 5.100*

*##2 69 5.067*

*##3 71 5.033*

*##4 87 5.000*

*##5 63 4.933*

*##6 89 4.900*

##### gapplyCollect

Like gapply, applies a function to each partition of a SparkDataFrame and collect the result back to R data.frame. The output of the function should be a data.frame. But, the schema is not required to be passed. Note that gapplyCollect can fail if the output of UDF run on all the partition cannot be pulled to the driver and fit in driver memory.

*# Determine six waiting times with the largest eruption time in minutes.*

result <- gapplyCollect(

df,

"waiting",

**function**(key, x) {

y <- data.frame(key, max(x$eruptions))

colnames(y) <- c("waiting", "max\_eruption")

y

})

head(result[order(result$max\_eruption, decreasing = **TRUE**), ])

*## waiting max\_eruption*

*##1 64 5.100*

*##2 69 5.067*

*##3 71 5.033*

*##4 87 5.000*

*##5 63 4.933*

*##6 89 4.900*

#### Run local R functions distributed using spark.lapply

##### spark.lapply

Similar to lapply in native R, spark.lapply runs a function over a list of elements and distributes the computations with Spark. Applies a function in a manner that is similar to doParallel or lapply to elements of a list. The results of all the computations should fit in a single machine. If that is not the case they can do something like df <- createDataFrame(list) and then use dapply

*# Perform distributed training of multiple models with spark.lapply. Here, we pass*

*# a read-only list of arguments which specifies family the generalized linear model should be.*

families <- c("gaussian", "poisson")

train <- **function**(family) {

model <- glm(Sepal.Length ~ Sepal.Width + Species, iris, family = family)

summary(model)

}

*# Return a list of model's summaries*

model.summaries <- spark.lapply(families, train)

*# Print the summary of each model*

print(model.summaries)

## Running SQL Queries from SparkR

A SparkDataFrame can also be registered as a temporary view in Spark SQL and that allows you to run SQL queries over its data. The sql function enables applications to run SQL queries programmatically and returns the result as a SparkDataFrame.

*# Load a JSON file*

people <- read.df("./examples/src/main/resources/people.json", "json")

*# Register this SparkDataFrame as a temporary view.*

createOrReplaceTempView(people, "people")

*# SQL statements can be run by using the sql method*

teenagers <- sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

head(teenagers)

*## name*

*##1 Justin*