MLlib + SparkR integration in Spark 1.5

This is publicly visible.

Authors: Xiangrui Meng
Revision History:

- July 2015: first version

Table of contents

Overview

This design doc discusses the MLlib + SparkR integration for Spark 1.5, including planned features, implementation ideas, and stages.

Requirements

In Spark 1.5, we want SparkR users to use MLlib to fit ML linear models to large-scale datasets, using the same syntax as in R's lm/glm/glmnet. We should produce the same result as R's lm/glm/glmnet to help R users migrate to SparkR. The features we are going to support are

- R formula with limited operators
- linear regression and logistic regression
- elastic-net regularization

Out of Scope

The following features are not targeted at Spark 1.5:

- model statistics (stretch goal)

- ML pipeline API
- use R packages to fit/select models in parallel
- other models like decision tree, naive bayes, k-means, etc
- cross validation
- weighted instances
- other error distributions (Gamma, Poisson)

Interfaces

References

R's lm (http://www.inside-r.org/r-doc/stats/lm):

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

R's glm (http://www.inside-r.org/r-doc/stats/glm):

```
glm(formula, family = gaussian, data, weights, subset,
    na.action, start = NULL, etastart, mustart, offset,
    control = list(...), model = TRUE, method = "glm.fit",
    x = FALSE, y = TRUE, contrasts = NULL, ...)
```

R's glmnet (http://www.inside-r.org/packages/cran/glmnet/docs/glmnet):

```
glmnet(x, y,
family=c("gaussian","binomial","poisson","multinomial","cox","mgaussian"),
    weights, offset=NULL, alpha = 1, nlambda = 100,
    lambda.min.ratio = ifelse(nobs<nvars,0.01,0.0001), lambda=NULL,
    standardize = TRUE, intercept=TRUE, thresh = 1e-07,  dfmax = nvars + 1,
    pmax = min(dfmax * 2+20, nvars), exclude, penalty.factor = rep(1, nvars),
    lower.limits=-Inf, upper.limits=Inf, maxit=100000,
    type.gaussian=ifelse(nvars<500,"covariance","naive"),
    type.logistic=c("Newton","modified.Newton"),
    standardize.response=FALSE, type.multinomial=c("ungrouped","grouped"))
```

(Note that glmnet doesn't provide the dataframe interface.)

SparkR's glm

The interface we provide is following R's lm, glm, and glmnet with limited options:

SparkR::glm(formula, family = c("gaussian", "binomial"), data, lambda = 0, alpha = 0, standardize = c(TRUE, FALSE))

- formula: a symbolic description of the model to be fitted
- family: error distribution. "gaussian" -> linear regression (least squares), "binomial" -> logistic regression
- data: SparkR DataFrame for training
- lambda: regularization parameter
- alpha: elastic-net mixing parameter (see glmnet's documentation for details)
- standardize: whether to standardize features before training

SparkR::glm returns a SparkR linear model object.

SparkR::predict(model, newData = NULL)

SparkR::predict returns a SparkR DataFrame with predictions.

SparkR::summary(model) [stretch goal]

SparkR::summary provides summary statistics of the model collected during training

Note that formula, family, and data are used in R's lm and glm, while lambda, alpha, and standardize are from glmnet. glm and glmnet support more error distributions in family. glmnet also support cross validation with a sequence of lambdas, which we are not going to support in Spark 1.5.

Implementation

R's model formula as a Transformer (SPARK-8774)

R's model formula

R's formula is used to describe a model. A detailed description can be found at the "Details" section of lm's documentation (http://www.inside-r.org/r-doc/stats/lm) and formula's doc (http://www.inside-r.org/r-doc/stats/formula). The basic operators are

- ~ | separate target and terms
- + | concat terms, "+ 0" means removing intercept
- : | interaction (multiplication for numeric values, or binarized categorical values)
- - | remove a term, "- 1" means removing intercept
- * | factor crossing
- ^ | factor crossing to a certain degree
- . | all columns except target

- math functions

Some simple examples:

- y ~ a + b means model y = w0 + w1*a + w2*b
- y ~ a + b + a:b - 1 means model y = w1*a + w2*b + w3*a*b, which is the same as y ~ (a + b)^2 + 0

If a column is a factor (string) column, glm will apply one-hot encoding first. For example, y ~ (a + gender)^2, where gender takes three values: male, female, and unknown, means y = w0 + w1*a + w2*gender$male + w3*gender$female + w4*a*gender$male + w5*a*gender$female. gender$male = 1 if gender is male, otherwise 0. gender$unknown doesn't show up in the model to avoid linear dependencies between features.

Formula as a Transformer

To support R's formula, we want to implement it as a Transformer under MLlib's pipeline API. Compared to native R implementation, the pros and cons are:

Pros

- Scala implementation that simplifies maintenance
- callable from Python, Java, and Scala
- leverage existing transformers in MLlib
- support Vector types (in the future)
- no R process on worker nodes

Cons

- full formula support might be tricky to implement

Scala interface:

```
class RFormula extends Transformer with HasFeaturesCol with HasLabelCol {
  val formula: Param[String] = …

  def setFormula(value: String): this.type = …

  ...
}
```

formula.transform(dataset) adds two new columns to the input DataFrame, by default, features: Vector and label: Double.

Scala example:

```
val formula = new RFormula()
  .setFormula("y ~ x + z")
```

```
val lr = new LogisticRegression()
val pipeline = new Pipeline()
  .setStages(Array(formula, lr))
val model = pipeline.fit(training)
```

Python example:

```
formula = RFormula(formula="y ~ x + z")
```

Formula parser

The core implementation would be a parser for R's formula. We could use Scala parser combinators to parse operators and terms. In Spark 1.5, we only want to implement basic formula support through a series of PRs:

1. ~, +
2. -, .
3. :, ^, *
4. factor column (stretch goal for 1.5)
5. poly, etc (stretch goal for 1.5)
6. vector support (1.6)
7. others

After the first PR, the rest of the work should be independent of the linear model support, which is discussed in the next section. Note that we should leverage on existing transformers in MLlib, e.g., StringIndexer, OneHotEncoder, PolynomialExpansion, and VectorAssembler. The formula should be parsed/compiled into an ML pipeline model.

R generates feature names for interactions and transformations. We should generate ML feature attributes similarly, but we don't need to implement the exact R heuristic. For example, for gender, R will generate "gendermale" and "genderfemale". We could use delimiters. The naming schema is not discussed in this doc, and we don't expect users to use those names programmatically.

SparkR::glm, predict, and summary

sparkr::glm should be implemented as a simple wrapper over the R formula transformer and the linear/logistic regression in MLlib. We are going to implement "GLMWrapper" under package "org.apache.spark.ml.api.r":

```
object GLMWrapper {
  def fit(formula: String, family: String, data: DataFrame, lambda: Double,
alpha: Double): GLMWrapperModel
```

```
  def predict(model: GLMWrapperModel, newData: DataFrame): DataFrame
}
```

where GLMWrapperModel is essential a PipelineModel consisting of RFormula and the fitted linear model with utility methods to extract the linear model coefficients, as well as a reference to the training dataset.

Pseudocode:

```
SparkR::glm(formula, family = c("gaussian", "binomial"), data, lambda = 0,
alpha = 0)

javaModel = callJMethod(
    "GLMWrapper.fit", formula, family, data@sdf, lambda, alpha)
model = # convert Java (pipeline) model to R model, or and keep the reference
to the Java model for prediction

SparkR::predict(model, newData = NULL)

callJMethod("GLMWrapper.fit", model@jmodel, newData@sdf)

SparkR::summary(model) [stretch goal]
```

Testing

We want to make sure we output the same result as R's glmnet. Note that glmnet's formulation is slightly different from R's lm and glm due to standardization. We use glmnet as reference.

Stages

The main blocker would be the implementation of RFormula. After than, the work could be distributed.

1. Implement RFormula with ~ and + support. (0.5w)
2. In parallel:
    a. implement -, . support and others mentioned in "Formula parser" (3w+, could be done in parallel)
    b. ML attribute (feature names) generation (0.5w)
    c. implement SparkR::glm and SparkR::predict in SparkR (1w)
    d. add RFormula wrapper in Python (1d)
    e. model statistics (stretch goal, depending on SPARK-8538 and SPARK-8539)