Rather than focusing on the details of logistic regression, we will focus more on how we can use R and some carefully written SQL statements to iteratively minimize a cost function. We will also use the condusco R package, which allows us to iterate through the results of a query easily.

**A Simple Logistic Regression Example**

Let’s start with a simple logistic regression example. We’ll simulate an outcome \(y\) based on the fact that \(Pr(y=1) = \frac{e^{\beta x}}{1+e^{\beta x}}\). Here \(\beta\) is a vector containing the coefficients we will later be estimating (including an intercept term). In the example below, our \(x\) values are uniform random values between -1 and 1.

set.seed(1)

# the number of samples

n <- 1000

# uniform random on (-1,1)

x1 <- 2\*runif(n)-1

x2 <- 2\*runif(n)-1

x <- cbind(1, x1, x2)

# our betas

beta <- c(-1, -3.0, 5.0)

probs <- exp(beta %\*% t(x))/(1+exp(beta %\*% t(x)))

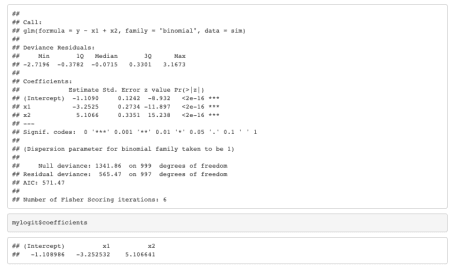
y <- rbinom(n,1,probs)

sim <- data.frame(id = seq(1:n), y = y, x1 = x1, x2 = x2)

mylogit <- glm(y ~ x1 + x2, data = sim, family = "binomial")

summary(mylogit)

mylogit$coefficients



As expected, the coefficients of our logistic model successfully approximate the parameters in our beta vector.

**In-database Logistic Regression**

Now, let’s see if we can find a way to calculate these same coefficients in-database. In this example, we’re going to use Google BigQuery as our database, and we’ll use condusco’s run\_pipeline\_gbq function to iteratively run the functions we define later on. To do this, we’ll need to take care of some initial housekeeping:

library(bigrquery)

library(whisker)

library(condusco)

# Uncomment and define your own config

# config <- list(

# project = '',

# dataset = '',

# table\_prefix = ''

# )

# a simple whisker.render helper function for our use-case

wr <- function(s, params=config){whisker.render(s,params)}

# put the simulated data in GBQ

insert\_upload\_job(

project = wr('{{{project}}}'),

dataset = wr('{{{dataset}}}'),

table = "logreg\_sim",

values = sim,

write\_disposition = "WRITE\_TRUNCATE"

)

Now, we’ll create the pipelines to do the logistic regression. Please note that the code below is quite verbose. While all of it is needed for the code to work, we’ll just focus on understanding how a couple of steps work. Once we understand one step, the rest is pretty easy.

First, we create a pipeline that does two things:

* create a main table containing all of our global settings
* calls another pipeline (log\_reg\_stack) with the global settings as inputs

Importantly, note that all of the parameters (eg. {{{project}}}) are dynamically swapped out in the query below with the wr function and the params variables. So this pipeline dynamically creates a query based on the parameters passed to it. We will call this pipeline later to run the process.

#

# Pipeline: log\_reg

#

log\_reg <- function(params){

print ("log\_reg")

query <- '

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_settings

AS

SELECT

"{{{project}}}" AS project,

"{{{dataset}}}" AS dataset,

"{{{data\_table}}}" AS data\_table,

{{{max\_steps}}} AS max\_steps,

{{{error\_tol}}} AS error\_tol,

{{{learning\_rate}}} AS learning\_rate,

"{{{id\_column}}}" AS id\_column,

"{{{label\_column}}}" AS label\_column,

"{{{fieldnames}}}" AS fieldnames,

"{{{constant\_id}}}" AS constant\_id,

"{{{table\_prefix}}}" AS table\_prefix

'

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

# Now run the log\_reg\_stack pipeline and pass the settings to it

invocation\_query <- '

SELECT \*

FROM {{{dataset}}}.{{table\_prefix}}\_settings

'

run\_pipeline\_gbq(

log\_reg\_stack,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

The above pipeline calls another pipeline, log\_reg\_stack, which is defined below. log\_reg\_stack creates a table with the field names that we will use in the logistic regression and then runs log\_reg\_stack\_field on each of the field names. Note that the invocation\_query below contains a query that results in one or more rows containing a field name. run\_pipeline\_gbq takes the results and iterates over them, calling log\_reg\_stack\_field on each one. Finally, it creates the \_labels table and calls log\_reg\_setup, passing it the results of the global settings query.

#

# Pipeline: stack variables

#

log\_reg\_stack <- function(params){

print ("log\_reg\_stack")

# Table: \_fieldnames

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_fieldnames

AS

SELECT TRIM(fieldname) AS fieldname

FROM (

SELECT split(fieldnames,',') AS fieldname

FROM (

SELECT '{{{fieldnames}}}' AS fieldnames

)

), UNNEST(fieldname) as fieldname

GROUP BY 1

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

# Run \_stack\_field

query <- "

DROP TABLE IF EXISTS {{{dataset}}}.{{{table\_prefix}}}\_stacked

"

tryCatch({

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)},

error = function(e){

print(e)

})

invocation\_query <- "

SELECT

a.fieldname AS fieldname,

b.\*

FROM (

SELECT fieldname

FROM {{{dataset}}}.{{{table\_prefix}}}\_fieldnames

GROUP BY fieldname

) a

CROSS JOIN (

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_settings

) b

"

run\_pipeline\_gbq(

log\_reg\_stack\_field,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

# Table: \_labels

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_labels

AS

SELECT

{{{id\_column}}} AS id,

{{{label\_column}}} AS label

FROM {{{data\_table}}}

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

# Run \_setup

invocation\_query <- "

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_settings

"

run\_pipeline\_gbq(

log\_reg\_setup,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

The log\_reg\_stack\_field and log\_reg\_setup pipelines are not particularly interesting. They do the groundwork needed to allow the log\_reg\_loop pipeline to iterate. The \_stacked table contains the feature names and their values, and the \_feature\_stats and features\_stacked\_vni tables contains normalized values used later. Finally, the \_fit\_params table contains the value of the fit parameters that will be updated as we iteratively minimize the cost function in the loop. The log\_reg\_setup pipeline ends by calling log\_reg\_loop, passing it the results of the global settings query.

log\_reg\_stack\_field <- function(params){

print ("log\_reg\_stack\_field")

destination\_table <- '{{{dataset}}}.{{{table\_prefix}}}\_stacked'

query <- "

SELECT {{{id\_column}}} AS id,

LTRIM('{{{fieldname}}}') AS feature\_name,

CAST({{{fieldname}}} AS FLOAT64) AS vi

FROM {{{data\_table}}}

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

destination\_table = wr(destination\_table, params),

use\_legacy\_sql = FALSE,

write\_disposition = 'WRITE\_APPEND',

create\_disposition = 'CREATE\_IF\_NEEDED'

)

}

log\_reg\_setup <- function(params){

print ("log\_reg\_setup")

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_feature\_stats

AS

SELECT feature\_name,

AVG(vi) AS mean,

STDDEV(vi) AS stddev

FROM {{{dataset}}}.{{{table\_prefix}}}\_stacked

GROUP BY feature\_name

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_features\_stacked\_vni

AS

SELECT

a.id AS id,

a.feature\_name AS feature\_name,

CASE

WHEN b.stddev > 0.0 THEN (vi - b.mean) / b.stddev

ELSE vi - b.mean

END AS vni

FROM {{{dataset}}}.{{{table\_prefix}}}\_stacked a

JOIN {{{dataset}}}.{{{table\_prefix}}}\_feature\_stats b

ON a.feature\_name = b.feature\_name

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

query <- "

INSERT INTO {{{dataset}}}.{{{table\_prefix}}}\_features\_stacked\_vni (id, feature\_name, vni)

SELECT

id,

'{{{constant\_id}}}' as feature\_name,

1.0 as vni

FROM {{{dataset}}}.{{{table\_prefix}}}\_stacked

GROUP BY 1,2,3

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

AS

SELECT

step,

param\_id,

param\_value,

cost,

stop,

message

FROM (

SELECT 1 as step,

feature\_name as param\_id,

0.0 as param\_value,

1e6 as cost,

false as stop,

'' as message

FROM {{{dataset}}}.{{{table\_prefix}}}\_stacked

GROUP BY param\_id

) UNION ALL (

SELECT 1 as step,

'{{{constant\_id}}}' as param\_id,

0.0 as param\_value,

1e6 as cost,

false as stop,

'' as message

)

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

# Run \_loop

invocation\_query <- "

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_settings

"

run\_pipeline\_gbq(

log\_reg\_loop,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

Next, we’ll create a loop pipeline that will iteratively calculate the cost function and update the \_fit\_params table with the latest update.

#

# Pipeline: loop

#

log\_reg\_loop <- function(params){

print ("log\_reg\_loop")

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_x\_dot\_beta\_i

AS

SELECT

a.id AS id,

SUM(a.vni \* b.param\_value) AS x\_dot\_beta\_i

FROM {{{dataset}}}.{{{table\_prefix}}}\_features\_stacked\_vni a

RIGHT JOIN (

SELECT param\_id, param\_value

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

WHERE STEP = (SELECT max(step) FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

) b

ON a.feature\_name = b.param\_id

GROUP BY 1

"

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

query <- '

INSERT INTO {{{dataset}}}.{{{table\_prefix}}}\_fit\_params (step, param\_id, param\_value, cost, stop, message)

SELECT

b.step + 1 as step,

b.param\_id as param\_id,

b.param\_value - {{{learning\_rate}}} \* err as param\_value,

-1.0 \* a.cost as cost,

CASE

WHEN ( abs((b.cost-(-1.0\*a.cost))/b.cost) < {{{error\_tol}}} ) OR (step+1 > {{{max\_steps}}})

THEN true

ELSE false

END AS stop,

CONCAT( "cost: ", CAST(abs((b.cost-(-1.0\*a.cost))/b.cost) AS STRING), " error\_tol: ", CAST({{{error\_tol}}} AS STRING)) as message

FROM (

SELECT

param\_id,

avg(err) as err,

avg(cost) as cost

FROM (

SELECT

a.id,

param\_id,

(1.0/(1.0 + EXP(-1.0 \* (c.x\_dot\_beta\_i))) - CAST(label AS FLOAT64)) \* vni as err,

CAST(label AS FLOAT64) \* LOG( 1.0/(1.0 + EXP(-1.0 \* (c.x\_dot\_beta\_i))) )

+ (1.0-CAST(label AS FLOAT64))\*(log(1.0 - (1.0/(1.0 + EXP(-1.0 \* (c.x\_dot\_beta\_i)))))) as cost

FROM (

SELECT a.id as id,

b.param\_id as param\_id,

a.vni as vni,

b.param\_value as param\_value

FROM {{{dataset}}}.{{{table\_prefix}}}\_features\_stacked\_vni a

JOIN (

SELECT param\_id, param\_value

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

WHERE STEP = (SELECT max(step) FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

) b

ON a.feature\_name = b.param\_id

GROUP BY 1,2,3,4

) a

JOIN {{{dataset}}}.{{{table\_prefix}}}\_labels b

ON a.id = b.id

JOIN {{{dataset}}}.{{{table\_prefix}}}\_x\_dot\_beta\_i c

ON a.id = c.id

)

GROUP BY param\_id

) a

JOIN (

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

WHERE STEP = (SELECT max(step) FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

) b

ON a.param\_id = b.param\_id

'

query\_exec(

project = wr('{{{project}}}', params),

query = wr(query, params),

use\_legacy\_sql = FALSE

)

# Loop or stop

query <- "

SELECT stop AS stop

FROM (

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

ORDER BY step DESC

LIMIT 1

)

"

res <- query\_exec(

wr(query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

if(res$stop == FALSE){

print("stop == FALSE")

invocation\_query <- '

SELECT \*

FROM {{{dataset}}}.{{table\_prefix}}\_settings

'

run\_pipeline\_gbq(

log\_reg\_loop,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

else {

print("stop == TRUE")

invocation\_query <- '

SELECT \*

FROM {{{dataset}}}.{{table\_prefix}}\_settings

'

run\_pipeline\_gbq(

log\_reg\_done,

wr(invocation\_query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

}

And finally, a log\_reg\_done pipeline that outputs the results:

#

# Pipeline: done

#

log\_reg\_done <- function(params){

print ("log\_reg\_done")

# Display results in norm'd coords

query <- '

SELECT "normalized coords parameters" as message,

step,

param\_id,

param\_value

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

WHERE step = (SELECT max(step) from {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

'

res <- query\_exec(

wr(query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

print(res)

# Display results in original coords

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_model\_params\_stacked

AS

SELECT

param\_id,

param\_value\_rescaled

FROM (

SELECT

a.param\_id AS param\_id,

a.param\_value + b.constant\_offset AS param\_value\_rescaled

FROM (

SELECT

step,

param\_id,

param\_value

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params

WHERE step = (SELECT max(step) from {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

AND param\_id = 'CONSTANT'

) a

JOIN (

SELECT

step,

'CONSTANT' as param\_id,

sum(-1.0\*param\_value\*mean/stddev) as constant\_offset

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params a

JOIN {{{dataset}}}.{{{table\_prefix}}}\_feature\_stats b

ON a.param\_id = b.feature\_name

WHERE step = (SELECT max(step) FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

GROUP BY 1,2

) b

ON a.param\_id = b.param\_id

) UNION ALL (

SELECT

param\_id,

param\_value/stddev as param\_value\_rescaled

FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params a

JOIN {{{dataset}}}.{{{table\_prefix}}}\_feature\_stats b

ON a.param\_id = b.feature\_name

WHERE step = (SELECT max(step) FROM {{{dataset}}}.{{{table\_prefix}}}\_fit\_params)

GROUP BY 1,2

)

"

res <- query\_exec(

wr(query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

print(res)

# transpose the \_model\_params\_stacked table

invocation\_query <- '

SELECT

a.list,

b.\*

FROM (

SELECT CONCAT("[", STRING\_AGG(CONCAT("{\\"val\\": \\"",TRIM(fieldname), "\\"}")), "]") AS list

FROM rstevenson.indb\_logreg\_001\_fieldnames

) a

CROSS JOIN (

SELECT \*

FROM rstevenson.indb\_logreg\_001\_settings

) b

'

run\_pipeline\_gbq(

log\_reg\_model\_params,

wr(invocation\_query, config),

wr('{{{project}}}', config),

use\_legacy\_sql = FALSE

)

print("DONE")

}

Our last pipeline, called at the end of the above pipeline, will transpose the stacked model params. In other words, it will output the parameters of the model in separate columns:

log\_reg\_model\_params <- function(params){

query <- "

CREATE OR REPLACE TABLE {{{dataset}}}.{{{table\_prefix}}}\_model\_params

AS

SELECT

{{#list}}

MAX(CASE WHEN param\_id='{{val}}' THEN param\_value\_rescaled END ) AS {{val}},

{{/list}}

MAX(CASE WHEN param\_id='{{constant\_id}}' THEN param\_value\_rescaled END ) AS {{constant\_id}}

FROM {{{dataset}}}.{{{table\_prefix}}}\_model\_params\_stacked

;"

res <- query\_exec(

wr(query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

print(res)

}

**Running the pipeline**

We are now ready to run the log\_reg pipeline. We’ll set up the invocation query with all of our global parameters. These will be stored in the \_settings table and then, after stacking and setup, the pipeline will iterate through the loop to calculate the logistic regression coefficients.

# Run the log\_reg pipeline with the following params (2D test)

invocation\_query <- '

SELECT

"{{{project}}}" as project,

"{{{dataset}}}" as dataset,

"{{{table\_prefix}}}" as table\_prefix,

"{{{dataset}}}.logreg\_sim" as data\_table,

"25" as max\_steps,

"1e-6" as error\_tol,

"6.0" as learning\_rate,

"id" as id\_column,

"y" as label\_column,

"x1, x2" as fieldnames,

"CONSTANT" as constant\_id

'

cat(wr(invocation\_query, config))

query\_exec(wr(invocation\_query, config), project=config$project, use\_legacy\_sql = FALSE)

run\_pipeline\_gbq(

log\_reg,

wr(invocation\_query, config),

project = wr('{{{project}}}', config),

use\_legacy\_sql = FALSE

)

After running the above, we should be able to query the table that holds the fitted parameters:

query <- "

SELECT \*

FROM {{{dataset}}}.{{{table\_prefix}}}\_model\_params

;"

query\_exec(

wr(query),

wr('{{{project}}}'),

use\_legacy\_sql = FALSE

)

As expected, these results are pretty close to our original beta values.

Please keep in mind that this is not ready to be released into the wild. Further improvements include modifications to deal with categorical variables, output describing whether a logistic fit is statistically significant for a particular parameter, and options for controlling step-sizes. But it does show the concept of how an iterative process like logistic regression can be done while using the database to maintain state.

**Prediction**

Now that we have fit the logistic regression model and the model is stored in the database, we can predict values using the model. We just need a prediction pipeline:

#

# Pipeline: predict

#

log\_reg\_predict <- function(params){

query <- '

SELECT

1/(1+exp(-1.0\*(CONSTANT + {{#list}}a.{{val}}\*b.{{val}} + {{/list}} + 0))) as probability

FROM {{{dataset}}}.{{{table\_prefix}}}\_model\_params a

CROSS JOIN {{{data\_table}}} b

ORDER BY {{{id\_column}}}

'

res <- query\_exec(

wr(query, params),

wr('{{{project}}}', params),

use\_legacy\_sql = FALSE

)

}

Note that the above uses whisker to calculate the dot product \(x\beta\) by expanding a JSON-formatted array of field names into {{#list}}a.{{val}}\*b.{{val}} + {{/list}} code. In the code below, we will create a JSON-formatted array of field names. Now let’s run the predictions:

# Run the prediction pipeline with the following params

invocation\_query <- '

SELECT

"{{{project}}}" as project,

"{{{dataset}}}" as dataset,

"{{{table\_prefix}}}" as table\_prefix,

"{{{dataset}}}.logreg\_sim" as data\_table,

"id" as id\_column,

CONCAT("[", STRING\_AGG(CONCAT("{\\"val\\": \\"",TRIM(fieldname), "\\"}")), "]") AS list

FROM {{{dataset}}}.{{{table\_prefix}}}\_fieldnames

'

predictions <- run\_pipeline\_gbq(

log\_reg\_predict,

wr(invocation\_query, config),

project = wr('{{{project}}}', config),

use\_legacy\_sql = FALSE

)

Let’s test the rounded predictions to see how well they approximate the outcomes:

# inspect first 5 true probs vs. predicted probabilities

head(probs[1:5])

head(predictions[[1]]$probability[1:5])

# mean relative error between true probs and predicted probabilities

mean((abs(probs-predictions[[1]]$probability))/probs)

Our model-based logistic regression model predicts the true probabilities with a mean relative error of about 7%.

**Next steps**

We have shown how to train and store a logistic regression model in a database. We can then predict outcomes given features that are also stored in the database without having to move data back and forth to a prediction server. In this particular example, it would likely be much faster to move the data to a computer and run the predictions there. However, certain use cases exist where in-database modeling could be an avenue for consideration. Further, since logistic models are fundamental to many types of tree and forest predictors, in-database logistic regression would be a necessary step in developing in-database tree methods. It remains to be seen if this approach can be easily translated into the tidyverse modeldb package.