# Scalable Data Analysis with Spark and Microsoft R Server

Ali Zaidi, alizaidi@microsoft.com

October 25th, 2016

# Scalable Data Analysis with R and Spark

# Unpacking the Title

- Scalable - should be able to accommodate large datasets
    - $p > n$
    - streaming datasets
    - generally large datasets where you want to do some inference
- R and Spark
    - R is great at statistical modeling, visualization and inference
    - R is also lazy, and functional
    - Let's reuse the API

# Background

- Studied statistics and machine learning at the University of Toronto
- Joined Microsoft through the Revolution Analytics acquistion last year
- Have used R for about 7 years now
- Hadoop for 2.5 years
- Spark for 1.5 years

# Strengths of R

## Where R Succeeds

- Expressive
- Open source
- Extendable – nearly 10,000 packages with functions to use, and that list continues to grow
- Focused on statistics and machine learning – utilized by academics and practitioners
- Advanced data structures and graphical capabilities
- Large user community, academics and industry
- It is designed by statisticians

# Weaknesses of R

## Where R Falls Short

- It is designed by statisticians
- Inefficient at element-by-element computations
- May make large demands on system resources, namely memory
- Data capacity limited by memory
- Single-threaded

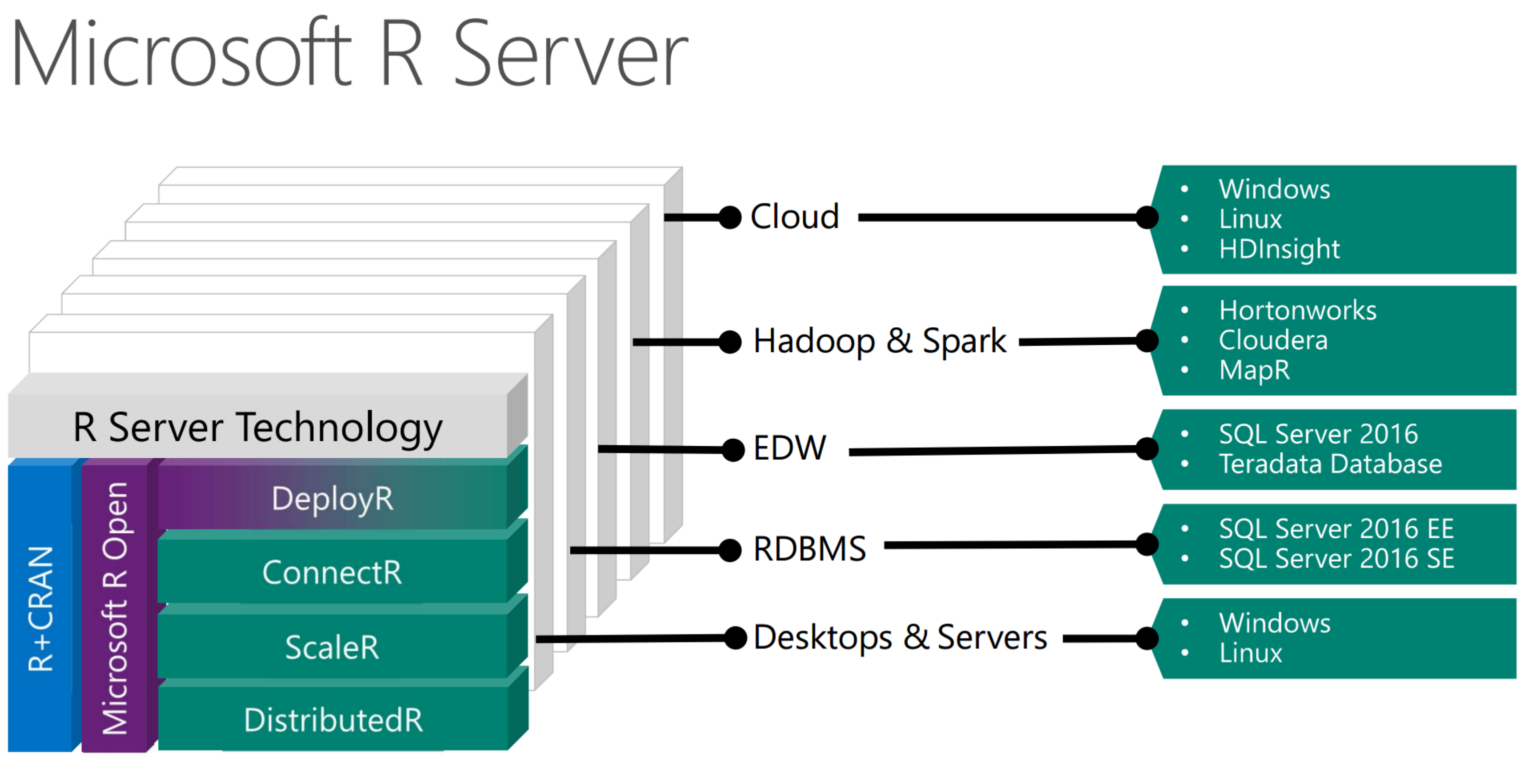
# Microsoft R Server - Scalable R

## WODA - Reusable API

- Breaks R's memory shackles by using external data frames (XDFs) and bridges to distributed systems (Spark, HadoopMR, Teradata)
- One of the core principles behind Microsoft R Server is the WODA framework
  - WODA – write once deploy anywhere
  - Encourages API reuse
- Provides seamless transition from a local environment to a cluster environment
- Don't want to rewrite the entire code-base in Java/Scala
- Deployment should be possible within seconds

# Microsoft R Server

## Components



# R APIs for Spark - A Tale of Three APIs

# The aRt of Being Lazy

## Lazy Evaluation in R

- R, like it's inspiration, Scheme, is a *functional* programming language
- R evaluates lazily, delaying evaluation until necessary, which can make it very flexible
- R has a high memory footprint, and can easily lead to crashes if you aren't careful

# R APIs for Spark

## `RxSpark`

- Allows the distribution of local R code into spark applications
- The only abstractions are those of the `RevoScaleR` package
  - data resides as distributed on-disk objects (`xdfd`'s)
  - Functions and data transformations are provided by traditional R objects
- Each `RevoScaleR` function invokes it's own Spark applications and converts XDFDs into RDDs/DataFrames
  - application persists for the duration of the job to avoid JVM creation overhead
  - for multi-iteration jobs, data is cached
  - no need for the developer to write any Spark code
- Available through Azure HDInsight Clusters
  - currently utilizing Spark 1.6
- User defined functions can be applied at the data partition level using `rxExec`

# R APIs for Spark

## `SparkR`

- The standard R API for Spark since 1.4 (MLLib support started in 1.5)
- R package provides functions that invoke functions directly on the JVM
  - Uses a RPC server and provides JVM wrappers
- `DataFrame` support is inspired by the `dplyr` package
  - tries to emulate `dplyr` syntax, but doesn't use NSE
  - `dplyr`: `taxi %>% group_by(pickup_nhood) %>% summarise(ave_delay = mean(pickup_delay))`
  - `SparkR`: `taxi %>% group_by(taxi$pickup_nhood) %>% summarise(ave_delay = mean(taxi$pickup_delay))`
- Limited ML: `glm`, `naive-bayes`, and `kmeans`
- Spark 2.0: support for custom UDFs using `dapply` and `gapply`
  - each partition must fit into an R process on the worker node

# R APIs for Spark

Masking with `dplyr`

# R APIs for Spark

## `sparklyr`

- Turns out `dplyr` already has a SQL backend
- Since all Spark DataFrame operations are conducted at the Spark SQL level, utilize `dplyr`'s SQL backend rather than JVM wrappers
- 100% support for `dplyr` SQL inside of Spark, including NSE:
  - `_lyr`: `taxi %>% group_by(pickup_nhood) %>% summarise(ave_delay = mean(pickup_delay))`
- Full support for all `SparkML`
- No UDF support yet

# R APIs for Spark

## `sparklyr` relies on RPC layer provided by `sparkapi`

- `sparklyr` is meant to be the DSL, providing easy data manipulation with `dplyr` and ML analogously to `stats` and other modeling packages
- The core RPC layer is not inside of `sparklyr`
- `sparkapi` provides the core R to Java RPC bridge publicly, and provides a simple extension mechanism to call arbitrary Spark APIs packages
  - e.g., extensions to connect o H20 Sparkling Water with `sparkapi` now exist

# Azure HDInsight

## Full Managed Hadoop/Spark on the Cloud

- With Azure HDInsight, you don't have to choose
- Can use any combination of the APIs for your data science application
- Focus less on optimizing code, rewriting your functions, and focus more on developing applications

# Data Manipulation Examples

# `sparklyr`

## Tidy Data Manipulation Using `dplyr` syntax

- For data scientists comfortable with R, using `sparklyr` requires no prior Spark knowledge
- `dplyr` statements are converted to SQL statements, sent to Catalyst and optimized

# Import Into Spark DataFrames

```
origins <- file.path("wasb://mrs-spark@alizaidi.blob.core.windows.net",
                     "user/RevoShare/alizaidi/Freddie/Acquisition")
library(sparklyr)
sc <- spark_connect("yarn-client")
freddie_origins <- spark_read_csv(sc,
                                  path = origins,
                                  name = 'freddie_origins',
                                  header = FALSE,
                                  delimiter = "|"
                                  )
```

# Using `dplyr`

- Now it's a Spark DataFrame
- It is also of class `tbl_sql`, so all `dplyr` methods are converted to `Spark SQL` statements and run on the spark application defined through the spark context `sc`

```
class(freddie_origins)
```

```
## [1] "tbl_spark" "tbl_sql"   "tbl_lazy"  "tbl"
```

```
library(dplyr)
freddie_origins %>% head
```

```
## Source:   query [?? x 25]
## Database: spark connection master=yarn-client app=sparklyr local=FALSE
##
##      V1     V2    V3     V4    V5    V6    V7    V8    V9   V10    V11
##   <chr>  <int> <chr>  <int> <int> <chr> <int> <chr> <dbl> <chr>  <int>
## 1   751 199910     N 202909    NA   000     1     O    71    20 180000
## 2   733 199909     N 202908 29540   000     1     O    51       116000
## 3   755 199905     N 202904 29540    30     1     O    95    38 138000
## 4   669 200206     N 202901    NA   000     1     O    80    33 162000
## 5   732 199904     N 202903 17140   000     1     O    25    10  53000
## 6   715 199904     N 202903 17140   000     1     O    67    35  91000
## # ... with 14 more variables: V12 <int>, V13 <dbl>, V14 <chr>, V15 <chr>,
## #   V16 <chr>, V17 <chr>, V18 <chr>, V19 <int>, V20 <chr>, V21 <chr>,
## #   V22 <int>, V23 <int>, V24 <chr>, V25 <chr>
```

# Renaming Columns

```
freddie_rename <- freddie_origins %>% rename(
                          credit_score = V1,
                          first_payment = V2,
                          first_home = V3,
                          maturity = V4,
                          msa = V5,
                          mi_perc = V6,
                          num_units = V7,
                          occ_status = V8,
                          cltv = V9,
                          dti = V10,
                          upb = V11,
                          ltv = V12,
                          orig_rate = V13,
                          channel = V14,
                          ppm = V15,
                          prod_type = V16,
                          state = V17,
                          prop_type = V18,
                          post_code = V19,
                          loan_number = V20,
                          loan_purpose = V21,
                          orig_term = V22,
                          num_borrowers = V23,
                          seller = V24,
                          servicer = V25
                          )
freddie_rename %>% head
```

```
## Source:   query [?? x 25]
## Database: spark connection master=yarn-client app=sparklyr local=FALSE
##
##   credit_score first_payment first_home maturity   msa mi_perc num_units
```

# Create Date Fields

The origination date is buried inside the loan number field. We will pick it out by indexing the loan number substring:

```
freddie_rename %>% select(loan_number)
```

```
## Source:   query [?? x 1]
## Database: spark connection master=yarn-client app=sparklyr local=FALSE
##
##     loan_number
##           <chr>
## 1  F199Q1000001
## 2  F199Q1000002
## 3  F199Q1000003
## 4  F199Q1000004
## 5  F199Q1000005
## 6  F199Q1000006
## 7  F199Q1000007
## 8  F199Q1000008
## 9  F199Q1000009
## 10 F199Q1000010
## # ... with more rows
```

# Substring Operations

```
freddie_rename <- freddie_rename %>%
  mutate(orig_date = substr(loan_number, 3, 4),
         year = as.numeric(substr(loan_number, 3, 2)))

freddie <- freddie_rename %>%
  mutate(orig_year = paste0(ifelse(year < 10, "200",
                                   ifelse(year > 16, "19",
                                          "20")), year))

freddie <- freddie %>%
  mutate(orig_year = substr(orig_year, 1, 4))

freddie %>% head
```

```
## Source:   query [?? x 28]
## Database: spark connection master=yarn-client app=sparklyr local=FALSE
##
##   credit_score first_payment first_home maturity   msa mi_perc num_units
##          <chr>         <int>      <chr>    <int> <int>   <chr>     <int>
## 1          751        199910          N   202909    NA     000         1
## 2          733        199909          N   202908 29540     000         1
## 3          755        199905          N   202904 29540      30         1
## 4          669        200206          N   202901    NA     000         1
## 5          732        199904          N   202903 17140     000         1
## 6          715        199904          N   202903 17140     000         1
## # ... with 21 more variables: occ_status <chr>, cltv <dbl>, dti <chr>,
## #   upb <int>, ltv <int>, orig_rate <dbl>, channel <chr>, ppm <chr>,
## #   prod_type <chr>, state <chr>, prop_type <chr>, post_code <int>,
## #   loan_number <chr>, loan_purpose <chr>, orig_term <int>,
## #   num_borrowers <int>, seller <chr>, servicer <chr>, orig_date <chr>,
## #   year <dbl>, orig_year <chr>
```

# Calculate Average Credit Score by Year

```
fico_year <- freddie %>% group_by(orig_year, state) %>%
  summarise(ave_fico = mean(credit_score)) %>% collect
fico_year %>% head
```

```
## Source: local data frame [6 x 3]
## Groups: orig_year [6]
##
##   orig_year state ave_fico
##       <chr> <chr>    <dbl>
## 1      2008    NM 730.7395
## 2      2012    WI 769.2008
## 3      2015    MI 752.5942
## 4      2014    AR 750.9074
## 5      2011    KS 763.1039
## 6      2006    WI 728.6378
```

# Summarize In a Function

```r
year_state_sum <- function(val = "credit_score") {

  library(lazyeval)

  year_state <- freddie %>% group_by(orig_year, state) %>%
    summarise_(sum_val = interp(~mean(var), var = as.name(val)))

  year_state <- year_state %>% collect

  names(year_state)[3] <- paste0("ave_", val)

  return(year_state)

}
```

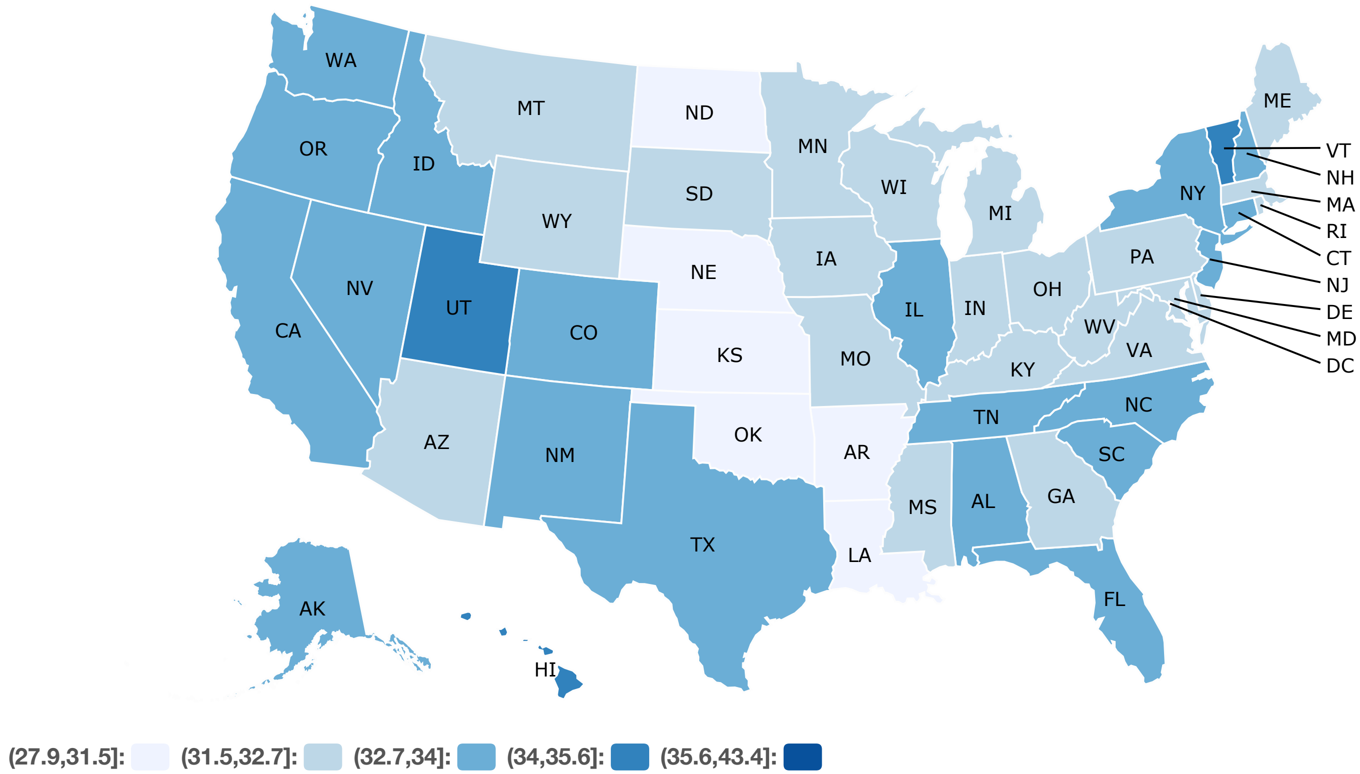
# Plot

```
library(rMaps)
year_state_sum("dti") %>%
  mutate(year = as.numeric(orig_year)) %>%
  rMaps::ichoropleth(ave_dti ~ state, data = .,
                     animate = "year",
                     geographyConfig = list(popupTemplate = "#!function(geo, data) {
                                    return '<div class=\"hoverinfo\"><strong>'+
                                    data.state + '<br>' + 'Average DTI in  '+ data.year + ': ' +
                                    data.ave_dti.toFixed(2)+
                                    '</strong></div>';}!#")) -> state_fico

state_fico$save("StateMapDTI.html", cdn = T)
```

# Plot

1999



WA
OR
CA
ID
NV
MT
WY
UT
AZ
CO
NM
ND
SD
NE
KS
OK
TX
MN
IA
MO
AR
LA
WI
IL
MI
IN
OH
KY
TN
MS
AL
GA
FL
SC
NC
VA
WV
PA
NY
ME
VT
NH
MA
RI
CT
NJ
DE
MD
DC
AK
HI
(27.9,31.5]:
(31.5,32.7]:
(32.7,34]:
(34,35.6]:
(35.6,43.4]:

# Machine Learning with `RxSpark`

# Predictive Models in the `RevoScaleR` package

### ETL

- Data import – Delimited, Fixed, SAS, SPSS, OBDC
- Variable creation & transformation
- Recode variables
- Factor variables
- Missing value handling
- Sort, Merge, Split
- Aggregate by category (means, sums)

### Descriptive Statistics

- Min / Max, Mean, Median (approx.)
- Quantiles (approx.)
- Standard Deviation
- Variance
- Correlation
- Covariance
- Sum of Squares (cross product matrix for set variables)
- Pairwise Cross tabs
- Risk Ratio & Odds Ratio
- Cross-Tabulation of Data (standard tables & long form)
- Marginal Summaries of Cross Tabulations

### Statistical Tests

- Chi Square Test
- Kendall Rank Correlation
- Fisher's Exact Test
- Student's t-Test

### Predictive Statistics

- Sum of Squares (cross product matrix for set variables)
- Multiple Linear Regression
- Generalized Linear Models (GLM) exponential family distributions: binomial, Gaussian, inverse Gaussian, Poisson, Tweedie. Standard link functions: cauchit, identity, log, logit, probit. User defined distributions & link functions.
- Covariance & Correlation Matrices
- Logistic Regression
- Predictions/scoring for models
- Residuals for all models

### Variable Selection

- Stepwise Regression

### Machine Learning

- Decision Trees
- Decision Forests
- Gradient Boosted Decision Trees
- Naïve Bayes

### Clustering

- K-Means

### Sampling

- Subsample (observations & variables)
- Random Sampling

### Simulation

- Simulation (e.g. Monte Carlo)
- Parallel Random Number Generation

### Custom Parallelization

- rxDataStep
- rxExec
- PEMA-R API

# Building Machine Learning Pipelines

## Functional Decompositions

```
estimate_model <- function(xdf_data = freddie[["train"]],
                           form = make_form(xdf_data, depVar = "default_flag"),
                           model = rxLogit, ...) {

  rx_model <- model(form, data = xdf_data, ...)

  return(rx_model)


}
```

# Executing Our Models In Parallel

## Functionals

```
computeContext <- RxSpark(consoleOutput=TRUE,
                          nameNode=myNameNode,
                          port=myPort,
                          executorCores=6,
                          executorMem = "10g",
                          executorOverheadMem = "5g",
                          persistentRun = TRUE

rxSetComputeContext(computeContext)

models <- list(rxLogit, rxDTree, rxDForest, rxBTrees)
trained_models <- rxExec(estimate_model, model = rxElemArg(models))
```

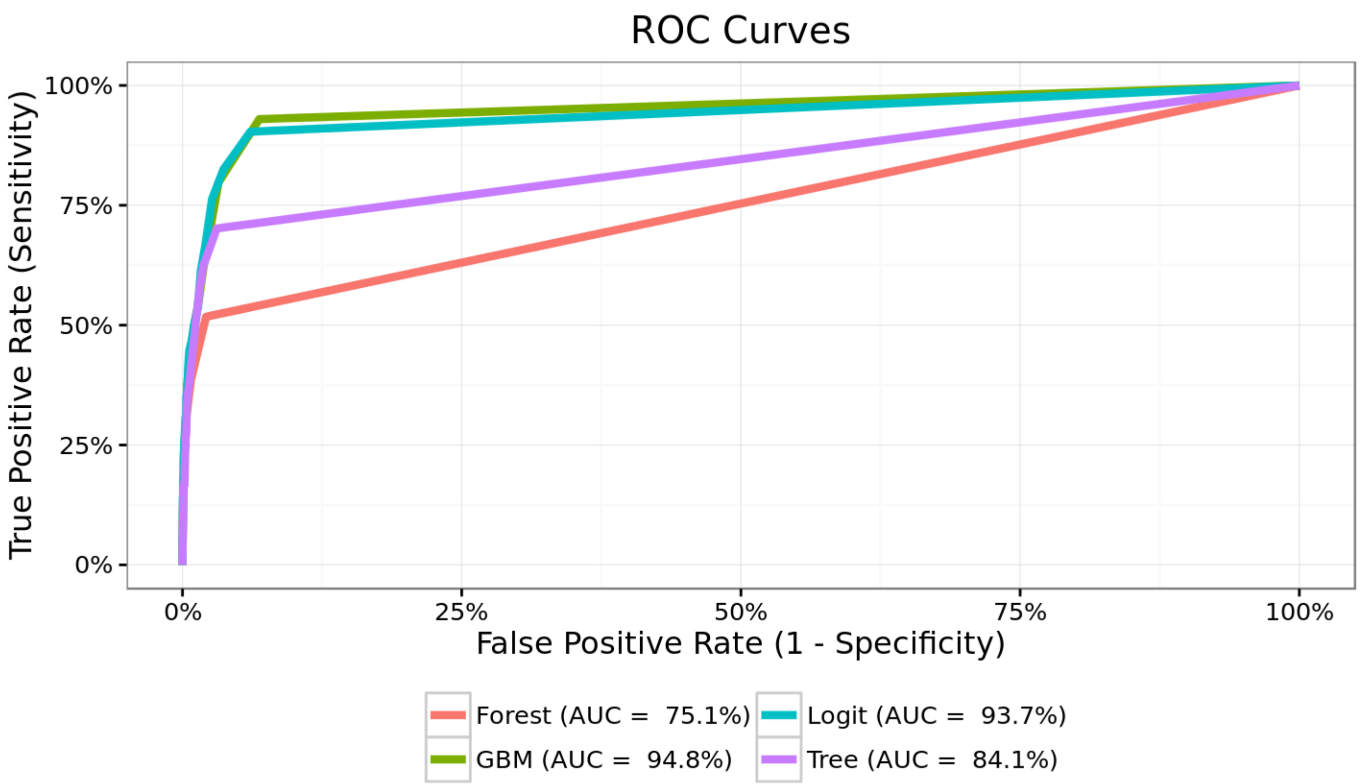
# Scoring the Models

```
default_pred <- function(model = default_model_gbm) {

  scored_xdf <- rxPredict(model,
                          mort_split$validate,
                          "scored.xdf")


  return(scored_xdf)
}
```

# Model Results



# What's Next

# Next Steps

- Sharing of Spark Contexts
  - currently, each of these APIs use their own backend to create Spark Sessions/Contexts, and cannot share objects across
- `RxSpark` will allow for using Spark DataFrames directly, and to share Spark sessions with other APIs
- Addition of new models and transformers in RevoScaleR

# Thank You!

# Resources

- https://github.com/akzaidi/R-cadence/tree/master/Spark
- https://bookdown.org/alizaidi/mrs-spark-ml/
- http://spark.rstudio.com/