SQL, Dataframes, and Julia

Tony Fraser : 10 September 2023

Assignment Overview

Utilize Visual Studio Code, Quarto markdown, SQL, and dataframes to navigate Julia fundamentals.

Conclusions

Amidst machine learning giants like TensorFlow and spark.ml, Julia distinguishes itself for tailored model designs. It's sleek, rapid, and bolstered by industry leaders like NVidia for GPU operations.

Julia Insights

1. **Positioned between Giants:** Julia sits between R and Python. Its syntax hints at Python, but the coding experience is reminiscent of R.
2. ***Blazing Speed:** Julia applications compile into fast and light C binaries. Their performance amplifies even further when tailored for specific GPUs.
3. **GTK Synergy:** In collaboration with GTK, Julia exhibits potential in software innovation — it could reshape tools like GIMP or Photoshop.
4. **Specialized Focus:** Julia excels in managing series and dataframes. Yet, areas like logging, markdown chunk processing, and comprehensive LaTeX support are secondary to its HPC features.
5. **Adoption Barriers:** Julia faces the same adoption challenges as Python and R. It is not object-oriented, presents a steep learning curve, and its integration into CI/CD pipelines can be intricate. It doesn't quite align with enterprise-friendly languages such as Object-Oriented Python, Scala, or C#.

Load The Castle Data Set

- Hosted on Kaggle, this is the European Castle data set.
- After this code block runs, data will be in both dataframes, and in the sqlite database.

Table headers:

- **CastleDetails**: Name, Latitude, Longitude, Adminstrative_Area_Level_1, Adminstrative_Area_Level_2, Municipality, Country, Postal_Code, Address, Editorial_Summary, Open_Hours, User_Rating, Wheelchair_Accessible_Entrance, Phone_Number, International_Phone_Number, Website
- **CastleUserReviews**:Name, Author, Rating, Review

```
using CSV, DataFrames, SQLite, HTTP, Gadfly

b = "https://raw.githubusercontent.com/tonythor/cuny-datascience/develop/data/eurocastles"
castle_details_content  = HTTP.get("$(b)/castle_details.csv").body
castle_user_reviews_content  = HTTP.get("$(b)/castle_user_reviews.csv").body

# castle_user_reviews_content <- comes back as a byte array of type ::Vector{UInt8}, to
# convert to a string do this: `content_string = String(castle_user_reviews_content)`

castles_df = CSV.File(IOBuffer(castle_details_content))      |> DataFrame
reviews_df = CSV.File(IOBuffer(castle_user_reviews_content)) |> DataFrame

# Create/Initalize/Populate SQLite DB
db = SQLite.DB("castles.db")
SQLite.load!(castles_df, db, "CastleDetails")
SQLite.load!(reviews_df, db, "CastleUserReviews")

function q(query::String, db::SQLite.DB)
    result = DBInterface.execute(db, query) |> DataFrame
    return result
end ; # <-- ugh this semicolon..
```

Run Some SQL

```
castles_query = """
  select Name, Municipality, Country, User_Rating
  from CastleDetails
  order by User_Rating, Country, Name
"""

ratings_query = """
  select Name, Rating
  from CastleUserReviews
  where Rating is not null
"""

castle_ratings_join_query = """
SELECT
    cd.Country,
    COUNT(DISTINCT cd.Name) AS TotalCastles,
    AVG(cr.Rating) as AvgUserRating
FROM
    CastleDetails cd
INNER JOIN
    CastleUserReviews cr ON cd.Name = cr.Name
WHERE
    cd.Country is not null
GROUP BY
    cd.Country
ORDER BY
    TotalCastles DESC, AvgUserRating DESC;

"""

cas = q(castles_query , db);
rat = q(ratings_query, db);
castle_ratings_join =q(castle_ratings_join_query, db);
```

A Quick Look At The Join Data

Note -> pretty tables is not as full featured as GT.

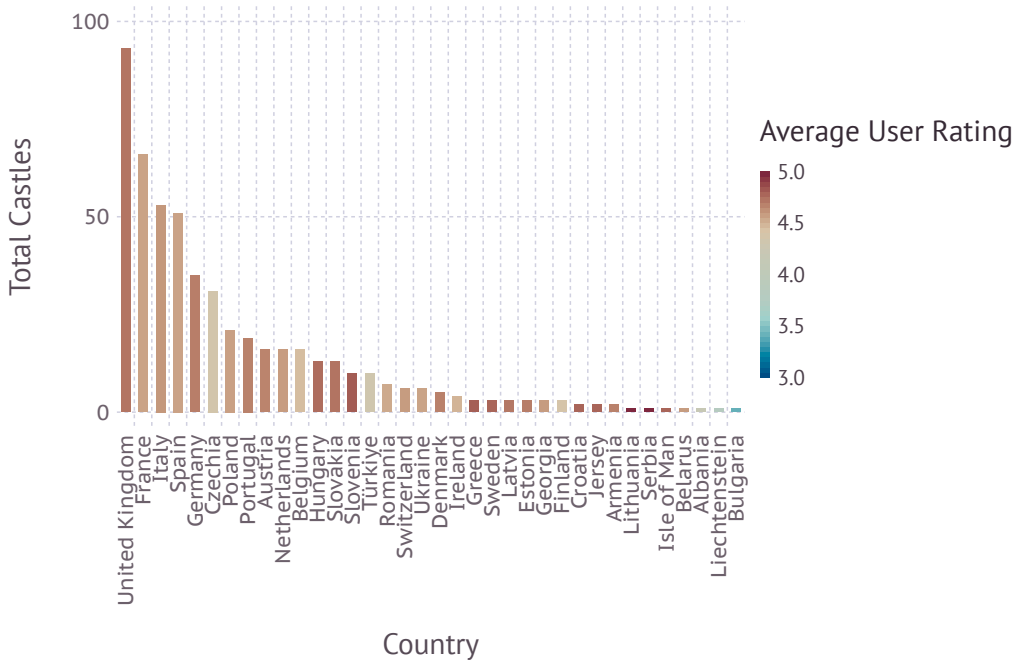
```
using PrettyTables
pretty_table(first(castle_ratings_join, 30), header=names(castle_ratings_join))
```

Country	TotalCastles	AvgUserRating
United Kingdom	93	4.74074
France	66	4.57099
Italy	53	4.61508
Spain	51	4.57317
Germany	35	4.70857
Czechia	31	4.3662
Poland	21	4.58163
Portugal	19	4.69663
Austria	16	4.6875
Netherlands	16	4.59722
Belgium	16	4.4625
Hungary	13	4.76923
Slovakia	13	4.74603
Slovenia	10	4.82979
Türkiye	10	4.32653
Romania	7	4.54286
Switzerland	6	4.6
Ukraine	6	4.56667
Denmark	5	4.7
Ireland	4	4.5
Greece	3	4.81818
Sweden	3	4.8
Latvia	3	4.73333
Estonia	3	4.71429
Georgia	3	4.6
Finland	3	4.4
Croatia	2	4.8
Jersey	2	4.8
Armenia	2	4.7
Lithuania	1	5.0

The Best Castles!

```
using Gadfly

p = plot(castle_ratings_join,
    x = :Country,
    y = :TotalCastles,
    color = :AvgUserRating,
    Geom.bar,
    Guide.xlabel("Country"),
    Guide.ylabel("Total Castles"),
    Guide.colorkey(title="Average User Rating"),
    Theme(bar_spacing=1mm)
)
display(p)
```



Cool Command Syntax!

```
using DataFrames
filter(row -> row[:Country] == "Italy", castle_ratings_join)      # string matching
filter(row -> row[:AvgUserRating] >4 , castle_ratings_join)       # numerical filtering
filter(row -> occursin("it", row[:Country]), castle_ratings_join) # country contains it

using DataFramesMeta
@where(castle_ratings_join, ismissing.(:Country))                 #<-- so cool!!
# @where(df, (:column1 .== "Value1") .| (:column2 .> 10))
# @where(df, in.(:column_name, Ref(["Value1", "Value2", "Value3"])))
# @where(df, occursin.("substring", :column_name))
# @where(df, (:column1 .> 10) .& (occursin.("substring", :column2)))

using Statistics
# Method 1: Fluent dataframe operations using anonymous functions
# This approach utilizes the |> operator to pass the result from one function to another.
# Anonymous functions (using the df -> ... syntax) provide the operations for each step.
joined_df1 = castles_df |>
           df -> innerjoin(df, reviews_df, on=:Name, makeunique=true) |>
           df -> filter(row -> !ismissing(row[:Country]), df) |>
           df -> combine(groupby(df, :Country),
                         nrow => :TotalCastles,
                         :Rating => mean => :AvgUserRating) |>
           df -> sort(df, [:TotalCastles, :AvgUserRating], rev=true)

# Method 2: Fluent dataframe operations using the Chain.jl package
# The Chain package provides the @chain macro that facilitates chaining operations
# in a more readable manner, especially for complex sequences of transformations.

using Chain
joined_df2 = @chain castles_df begin
    innerjoin(_, reviews_df, on=:Name, makeunique=true)
    filter(row -> !ismissing(row[:Country]), _)
    combine(groupby(_, :Country),
            nrow => :TotalCastles,
            :Rating => mean => :AvgUserRating)
    sort(_, [:TotalCastles, :AvgUserRating], rev=true)
end
```

What's Missing?

- **Speed Tests:** One of Julia's primary selling points is its speed, especially when compared to interpreted languages like R and Python. Though there are plenty of benchmarks online, real world experience with real world tasks could help form opinions
- **A GPU Demonstration:** Julia's seamless GPU integration is a game-changer for heavy computational tasks. Demonstrating the conversion of a simple matrix function into native GPU code, and then benchmarking it, would be fascinating.
- **GTK and Julia - A Simple Application:** Julia isn't just about number crunching. With its integration capabilities with GUI frameworks like GTK, for lightweight scientific type problems, it might be a plausible development platform.