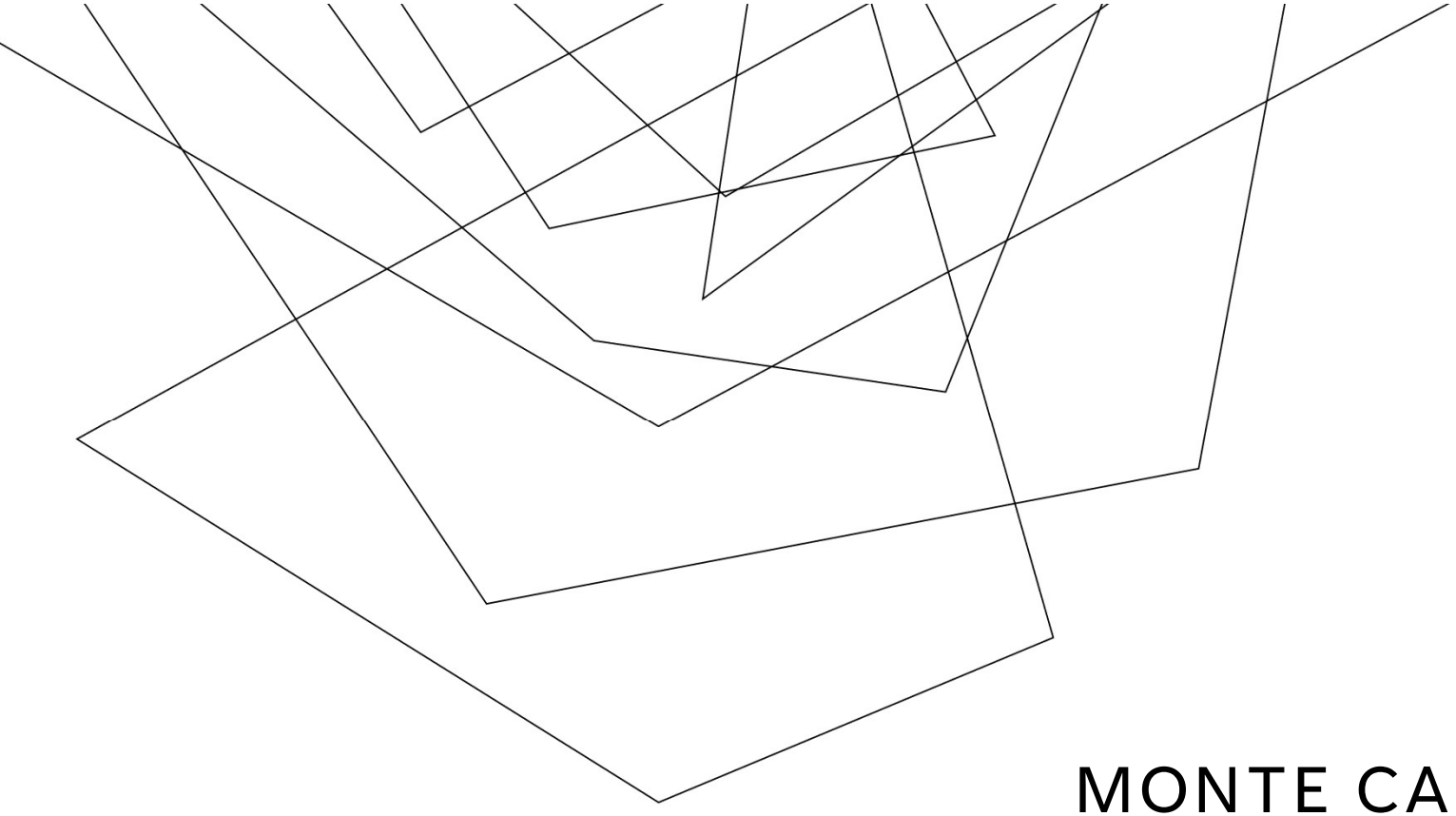


# MONTE CARLO THE PROJECT THAT SANK

Devin Larson

# AGENDA

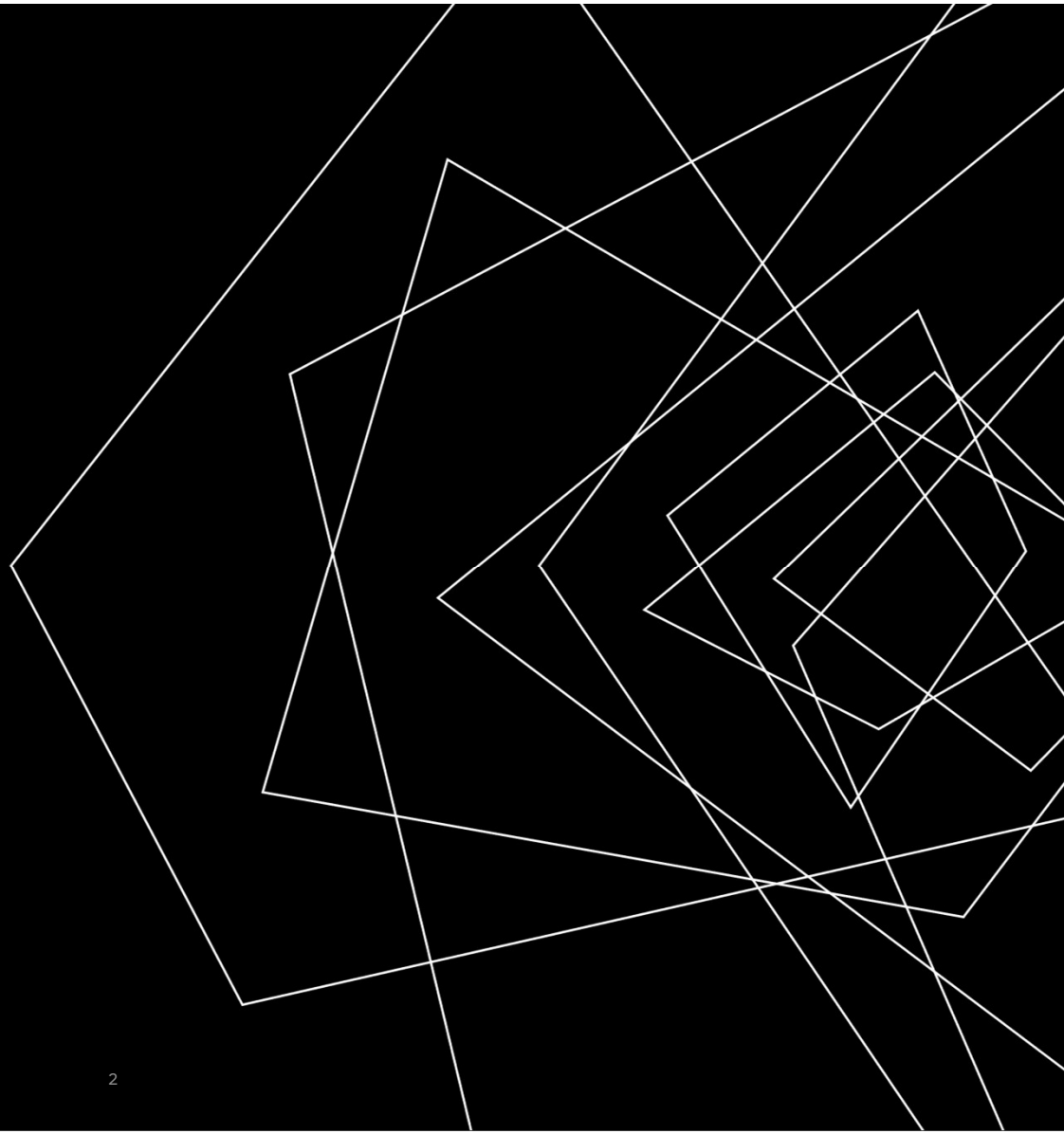
Math Background

Approach

Kernel's Deep Dive

Modeling

Fun



# INTRODUCTION TO MONTE CARLO

- A computational algorithm that relies on repeated random sampling to obtain numerical results.

- **Step 1:** Define a domain of possible inputs.
- **Step 2:** Generate inputs randomly from a probability distribution over the domain.
- **Step 3:** Perform a deterministic computation on the inputs.
- **Step 4:** Aggregate the results.

# PI EXAMPLE

Example Code(in R)

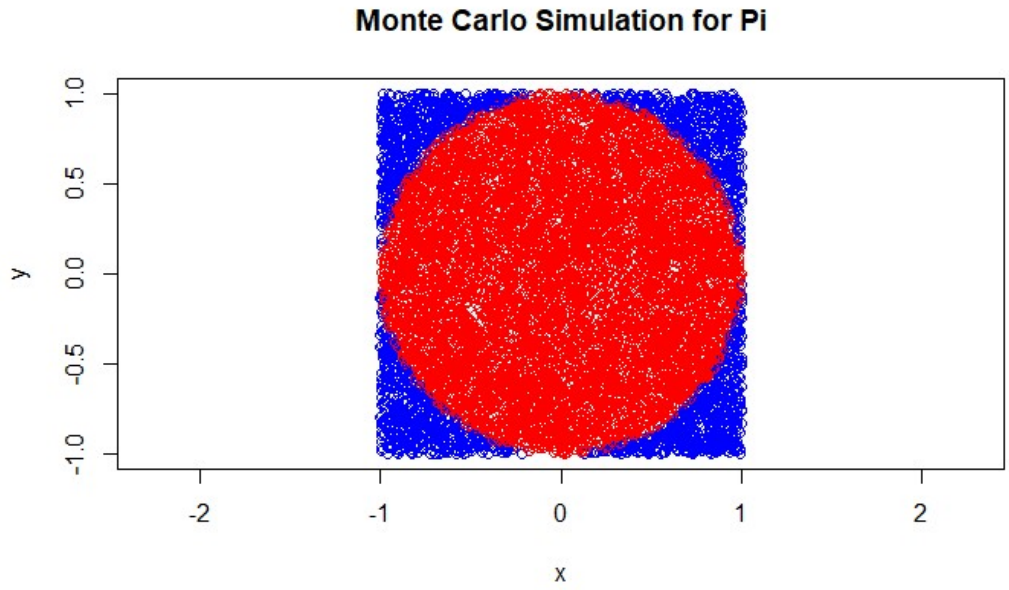
```r
# Monte Carlo Simulation in R to Estimate Pi

# Number of random points
num_points <- 10000

# Generate random points
x <- runif(num_points, min=-1, max=1)
y <- runif(num_points, min=-1, max=1)

# Calculate the number of points inside the circle
points_inside <- sum(x^2 + y^2 <= 1)

# Estimate Pi
pi_estimate <- (points_inside / num_points) * 4
```




Estimated value of Pi: 3.1236

# PROCESS

**1st Attempt**

CPU IMPLMENTATION in Python

- not scale able to the level I wanted
- ability to carry models was nice

**Python Approach**

Limited to certain applications hard to debug the GPU side due to more control being taken away through abstraction

**Using Python to Call CUDA Kernels**

Difficulty in debugging gave up to just use CUDA

**CUDA**

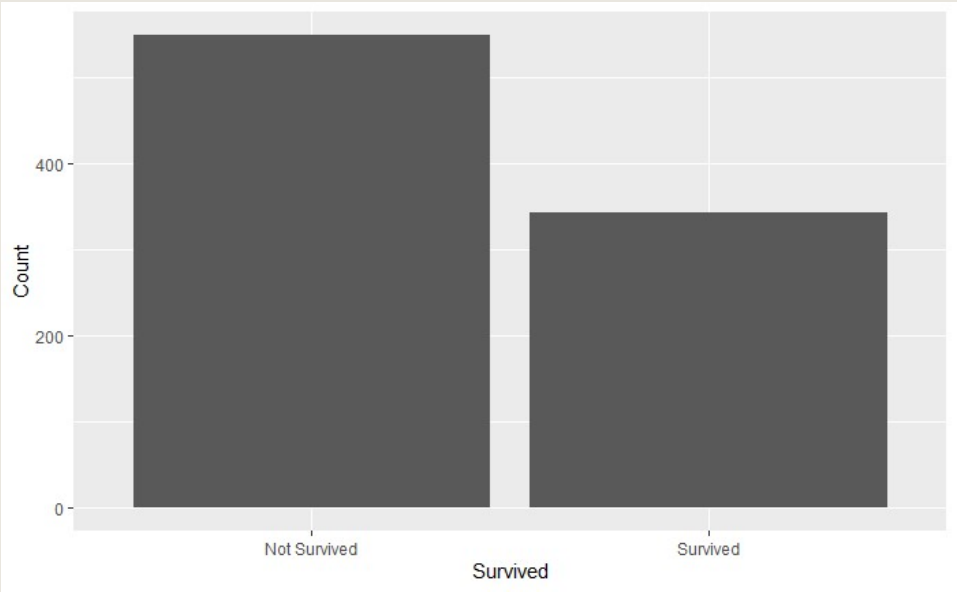
- a lot of time waisted trying other ways to just use CUDA due to the most control over what I am doing

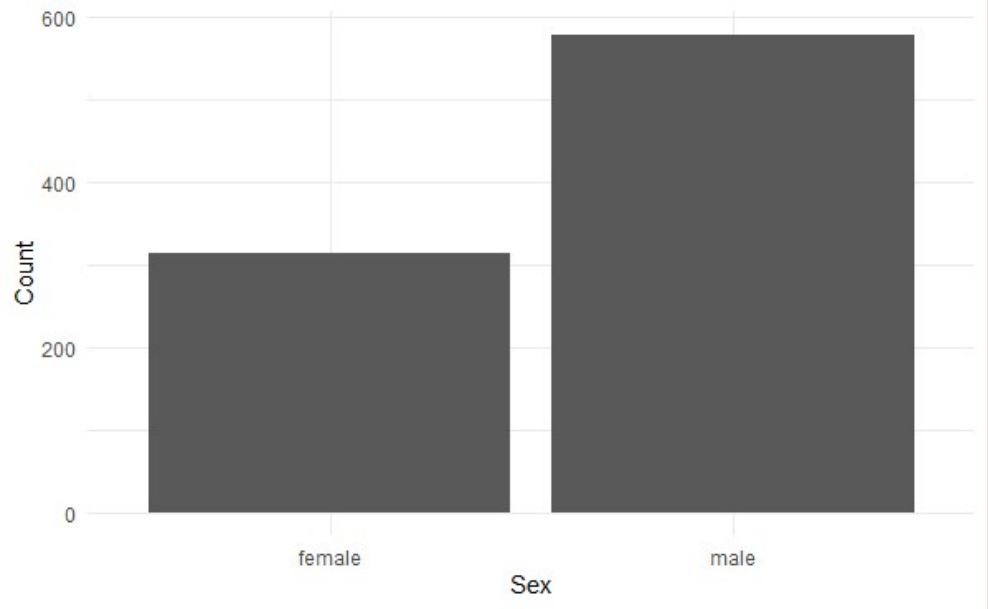
# DATA VARIABLES(892 PEOPLE REPRESENTED)

- `Survived` is the target variable we are trying to predict (**0** or **1**):
  - **1 = Survived**
  - **0 = Not Survived**
- `Pclass` (Passenger Class) is the socio-economic status of the passenger and it is a categorical ordinal feature which has **3** unique values (**1**, **2** or **3**):
  - **1 = Upper Class**
  - **2 = Middle Class**
  - **3 = Lower Class**
- ~~`Name`~~, Sex and Age are self-explanatory
- `SibSp` is the total number of the passengers' siblings and spouse
- `Parch` is the total number of the passengers' parents and children
- ~~`Ticke`~~`t` is the ticket number of the passenger
- `Fare` is the passenger fare
- `Embarked` is port of embarkation and it is a categorical feature which has **3** unique values (**C**, **Q** or **S**):
  - **C = Cherbourg**
  - **Q = Queenstown**
  - **S = Southampton**

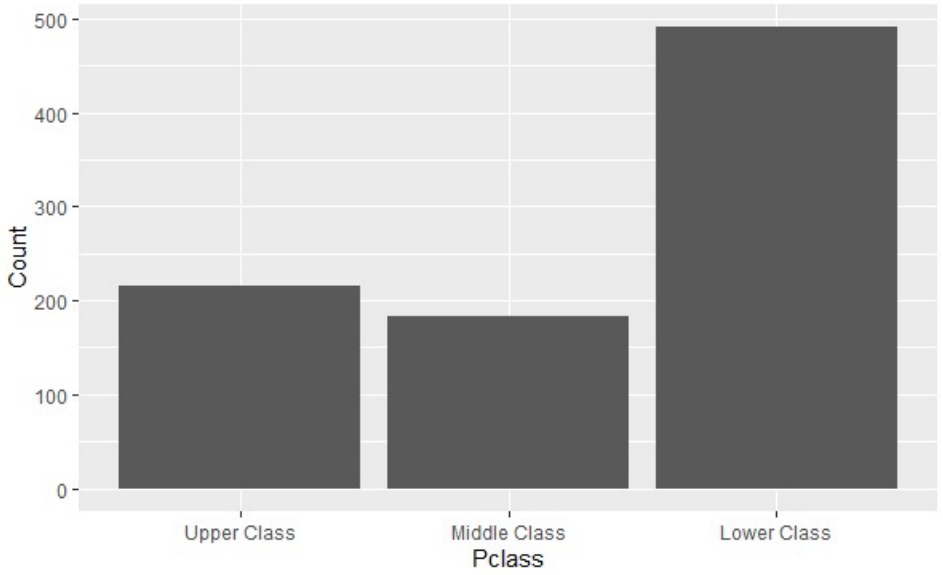
| PassengerId | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22 | 1 | 0 | A/5 21171 | 7.25 | | S |
| 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Thayer) | female | 38 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26 | 0 | 0 | STON/O2. | 7.925 | | S |
| 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35 | 1 | 0 | 113803 | 53.1 | C123 | S |
| 5 | 0 | 3 | Allen, Mr. William Henry | male | 35 | 0 | 0 | 373450 | 8.05 | | S |
| 6 | 0 | 3 | Moran, Mr. James | male | | 0 | 0 | 330877 | 8.4583 | | Q |
| 7 | 0 | 1 | McCarthy, Mr. Timothy J | male | 54 | 0 | 0 | 17463 | 51.8625 | E46 | S |
| 8 | 0 | 3 | Palsson, Master. Gosta Leonard | male | 2 | 3 | 1 | 349909 | 21.075 | | S |
| 9 | 1 | 3 | Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg) | female | 27 | 0 | 2 | 347742 | 11.1333 | | S |
| 10 | 1 | 2 | Nasser, Mrs. Nicholas (Adele Achem) | female | 14 | 1 | 0 | 237736 | 30.0708 | | C |
| 11 | 1 | 3 | Sandstrom, Miss. Marguerite Rut | female | 4 | 1 | 1 | PP 9549 | 16.7 | G6 | S |
| 12 | 1 | 1 | Bonnell, Miss. Elizabeth | female | 58 | 0 | 0 | 113783 | 26.55 | C103 | S |

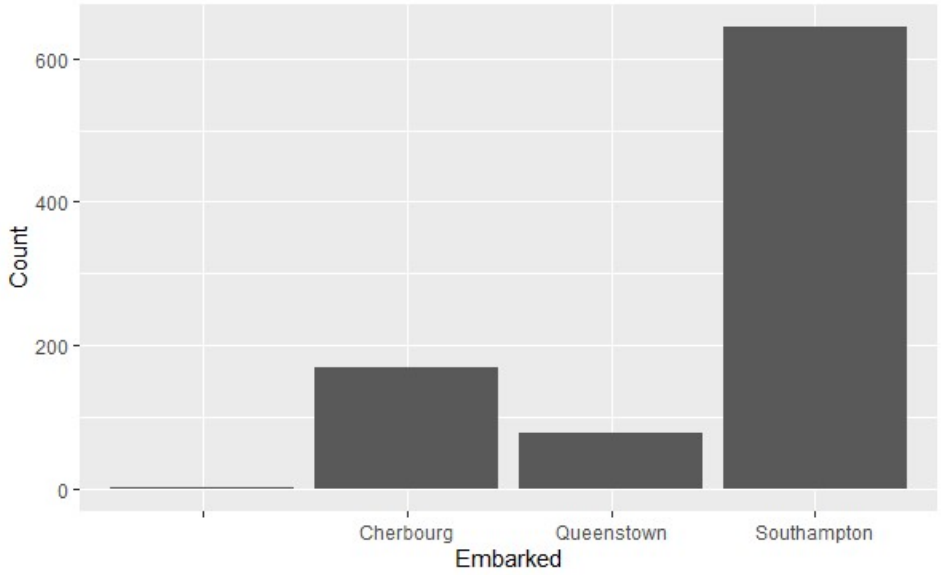
# CATEGORICAL DISTRIBUTIONS(2)



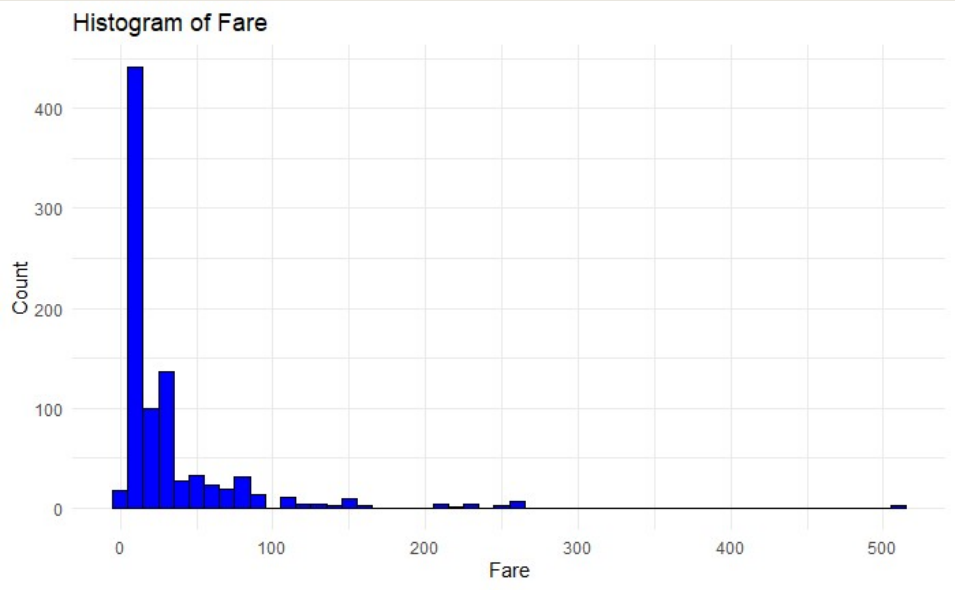



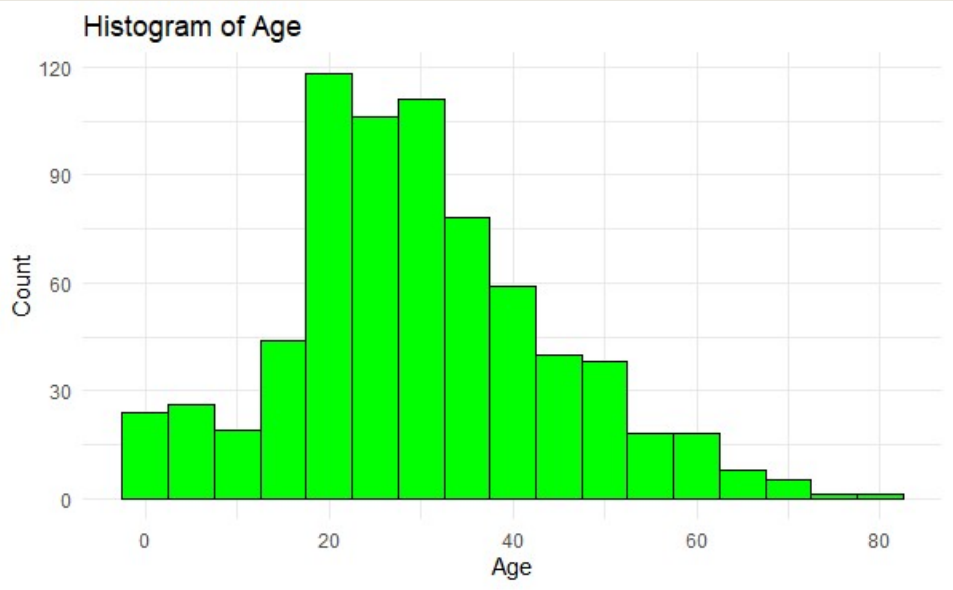
# CATEGORICAL DISTRIBUTIONS(3)



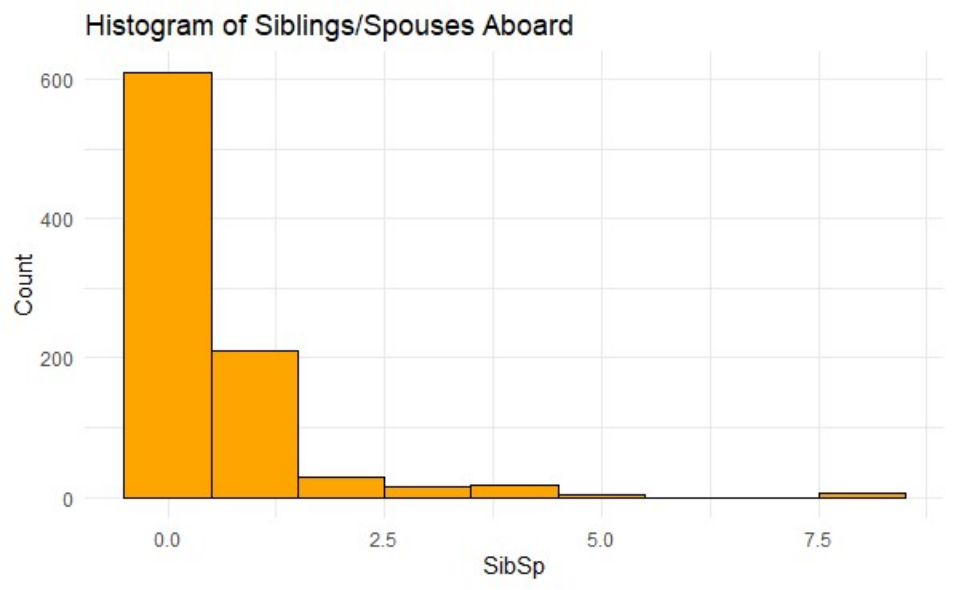



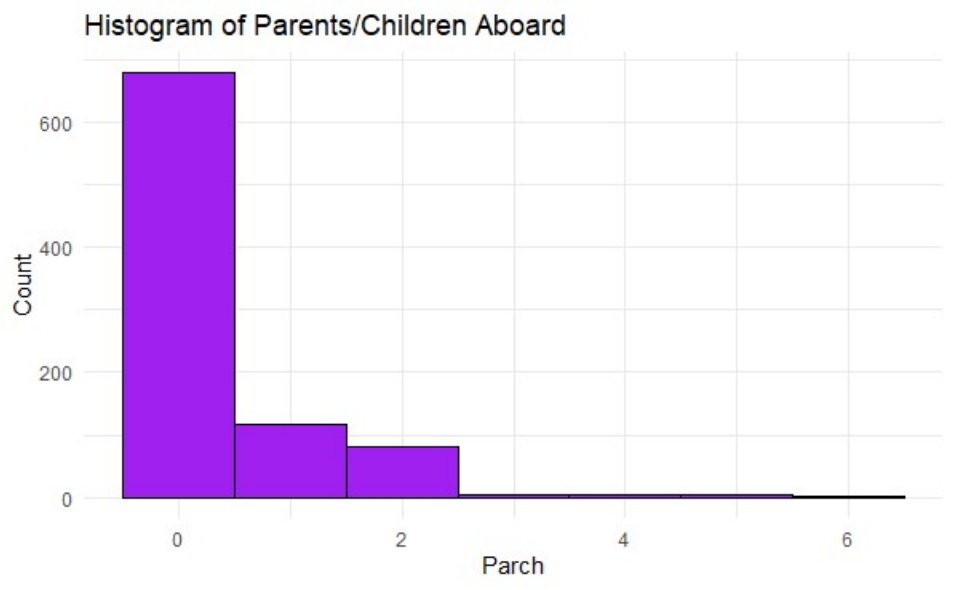
# LOG- NORMAL DISTRIBUTIONS



# NORMAL DISTRIBUTIONS



Side Note All Profiling was done for n = 1 million

KERNELS

Init

Generate Categorical 3

Generate Categorical 2

Generate Log Normal

Generate Normal

Rounding

# INIT

```
__global__ void init(curandState_t* states, unsigned long seed, int n) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        curand_init(seed, idx, 0, &states[idx]);
    }
}
```

- **Scalability**: Designed for scalability across many threads.
- **Reproducibility**: Consistent initialization allows for reproducible results in simulations.
- **GPU Only**: States never leaves the GPU till it is a variables final form

# INIT PROFLING

Initialization Time: 0 ms
Kernel Execution Time: 28.208128 ms
Memory Transfer Time: 0.002048 ms

# CATEGORICAL 3

```
//MINMIZE BRANCH DIVERGENCE
__global__ void generateCategorical3(curandState_t* states, int* results, int n, float ratio1, float ratio2) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= n) return;

    float randVal = curand_uniform(&states[idx]);
    float combinedRatio = ratio1 + ratio2; // Pre-calculate the sum of the ratios

    // Minimize branch divergence
    int category = 3; // Default category
    category = (randVal < combinedRatio) ? ((randVal < ratio1) ? 1 : 2) : category;

    results[idx] = category;
}
```

- Generates a uniform random value between 0 and 1 for each trial.
- Assigns a category based on the random value and predefined ratios:
- Reduced Overhead: Direct computation within GPU, minimizing data transfer between CPU and GPU.
- Optimized for Large n: Efficient for large-scale simulations due to parallel execution.

# CATEGORICAL 3 PROFILING

Initialization Time: 28.095488 ms
Kernel Execution Time: 0.002048 ms
Memory Transfer Time: 0.360448 ms

Time for generateCategorical3: 0.355328 ms
Data transferred for generateCategorical3: 4000000 bytes
Bandwidth for generateCategorical3: 10735.707031 MB/s

# CATEGORICAL 2

```cpp
__global__ void generateCategorical2(curandState_t* states, int* results, int n, float ratio) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= n) return;

    results[idx] = (curand_uniform(&states[idx]) < ratio) ? 0 : 1;
}
```

# CATEGORICAL 2 PROFILING

Initialization Time: 28.098560 ms
Kernel Execution Time: 0.361472 ms
Memory Transfer Time: 2.819296 ms

Time for generateCategorical2: 0.357408 ms
Data transferred: 4000000 bytes
Bandwidth: 10673.228516 MB/s

# NORMAL

```
__constant__ float TWO_PI = 2.0f * M_PI;
__global__ void generateNormal(curandState_t* states, float* results, int n, float mean, float stddev, int decimalPlaces) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= n) return;

    curandState localState = states[idx];
    float u1 = curand_uniform(&localState);
    float u2 = curand_uniform(&localState);

    // Using FMA for efficiency
    float radius = sqrtf(-2.0f * logf(u1));
    float angle = TWO_PI * M_PI * u2;
    float normalValue = fma(radius, cosf(angle), mean);
    normalValue = fma(normalValue, stddev, 0.0f);

    results[idx] = roundToDecimal(normalValue, decimalPlaces);
    states[idx] = localState;
}
```

- Utilizes the CURAND library for generating uniform random numbers: curand_uniform(&localState).
- Generates two uniform random numbers u1 and u2.
- Applies the Box-Muller formula to convert uniform random numbers to a normal distribution:
- float normalValue = mean + stddev * sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
- rounds

# NORMAL PROFILING

Initialization Time: 28.228607 ms
Kernel Execution Time: 0.402432 ms
Memory Transfer Time: 2.881728 ms

Time for normal_transform: 0.400384 ms
Data transferred for normal_transform: 4000000 bytes
Bandwidth for normal_transform: 9527.596680 MB/s

# LOG NORMAL

```cuda
__global__ void generateLognormal(curandState_t* states, float* results, int n, float mean, float stddev, int decimalPlaces) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx >= n) return;

    float z = curand_normal(&states[idx]);
    float lognormal = expf(fma(stddev, z, mean)); // Using FMA and expf for efficiency
    results[idx] = roundToDecimal(lognormal, decimalPlaces);
}
```

- Generates a normally distributed random number z using CURAND: float z = curand_normal(&states[idx]);
- Adjusts z to have the specified mean and standard deviation: z = stddev * z + mean;
- Applies the exponential function to convert the normal value to a lognormal distribution: float lognormal = exp(z);
- Rounds
- Maintains CURAND states within the kernel, reducing the need for additional memory operations.
- Must got back and calculate for the underlying normal distribution to set correct paramaters

```python
# Desired mean and standard deviation of the log-normal distribution
desired_mean = 29.82596385
desired_std = 38.06890240954017

# Solve for the parameters of the underlying normal distribution
sigma = np.sqrt(np.log((desired_std / desired_mean) ** 2 + 1))
mu = np.log(desired_mean) - sigma ** 2 / 2

# These mu and sigma can be used in your CUDA kernel
print("mu:", mu, "sigma:", sigma)
```

# LOG NORMAL PROFILING

Initialization Time: 28.150784 ms
Kernel Execution Time: 0.444416 ms
Memory Transfer Time: 2.949568 ms

Time for generateLognormal: 0.430080 ms
Data transferred: 4000000 bytes
Bandwidth: 8869.739258 MB/s

# ROUNDING

```
//IS NOT THE BEST FOR LOG WSAS WORSE FOR LOG BUT HAD MORE BETTER FOR NORAML
__device__ inline float roundToDecimal(float value, int decimalPlaces) {
    float scale = powf(10.0f, decimalPlaces); // Efficient scaling using powf
    return roundf(value * scale) / scale; // Round and scale back
}
```

- **Scaling the Value:** The function first calculates a scaling factor using powf(10.0f, decimalPlaces). This effectively creates a multiplier to scale the original number. For example, if decimalPlaces is 2, the scale factor will be 100.
- **Rounding the Scaled Value:** The original value is then multiplied by this scale factor, moving the decimal point to the right by decimalPlaces positions. The roundf function is then used to round this scaled value to the nearest integer.
- **Scaling Back**: After rounding, the function scales the value back to its original range by dividing it by the scale factor. This step restores the number to its original range but with the desired number of decimal places.

# ROUNDING PROFILING

- Roughly **0.006144ms** spent running
- Inline was a better choice for Normal that over the slow down in log normal

# GENERAL PROFILING

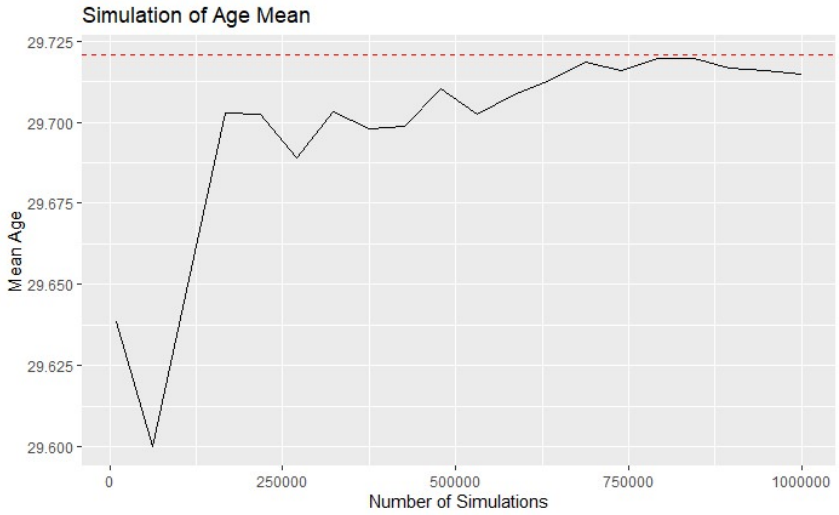
Even Comparison

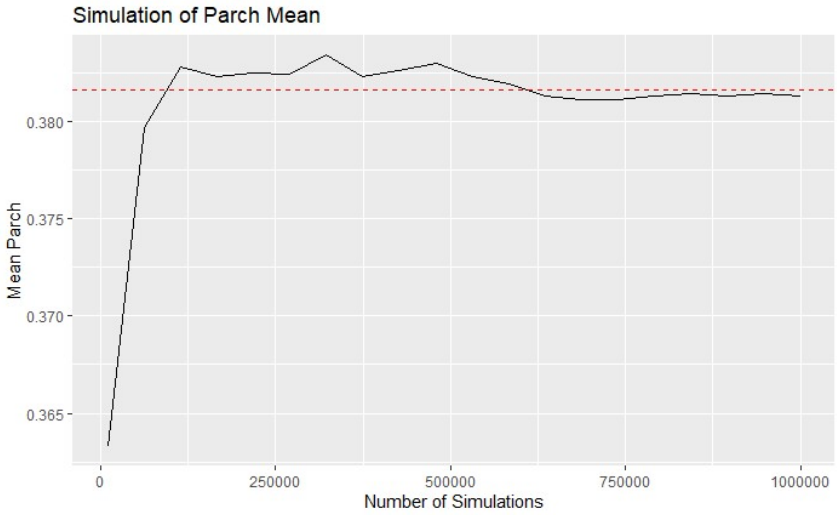
```
==3010== NVPROF is profiling process 3010, command: ./data 6 1000000
==3010== Profiling application: ./data 6 1000000
==3010== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.22%  84.298ms         3  28.099ms  28.015ms  28.250ms  init(curandStateXORWOW*, unsigned long, int)
                    0.51%  435.12us         1  435.12us  435.12us  435.12us  generateLognormal(curandStateXORWOW*, float*, int, float, float, int
                    0.46%  394.83us         1  394.83us  394.83us  394.83us  generateNormal(curandStateXORWOW*, float*, int, float, float, int)
                    0.41%  349.71us         1  349.71us  349.71us  349.71us  generateCategorical2(curandStateXORWOW*, int*, int, float)
                    0.40%  346.89us         1  346.89us  346.89us  346.89us  generateCategorical3(curandStateXORWOW*, int*, int, float, float)
```

Actual Execution Comparison

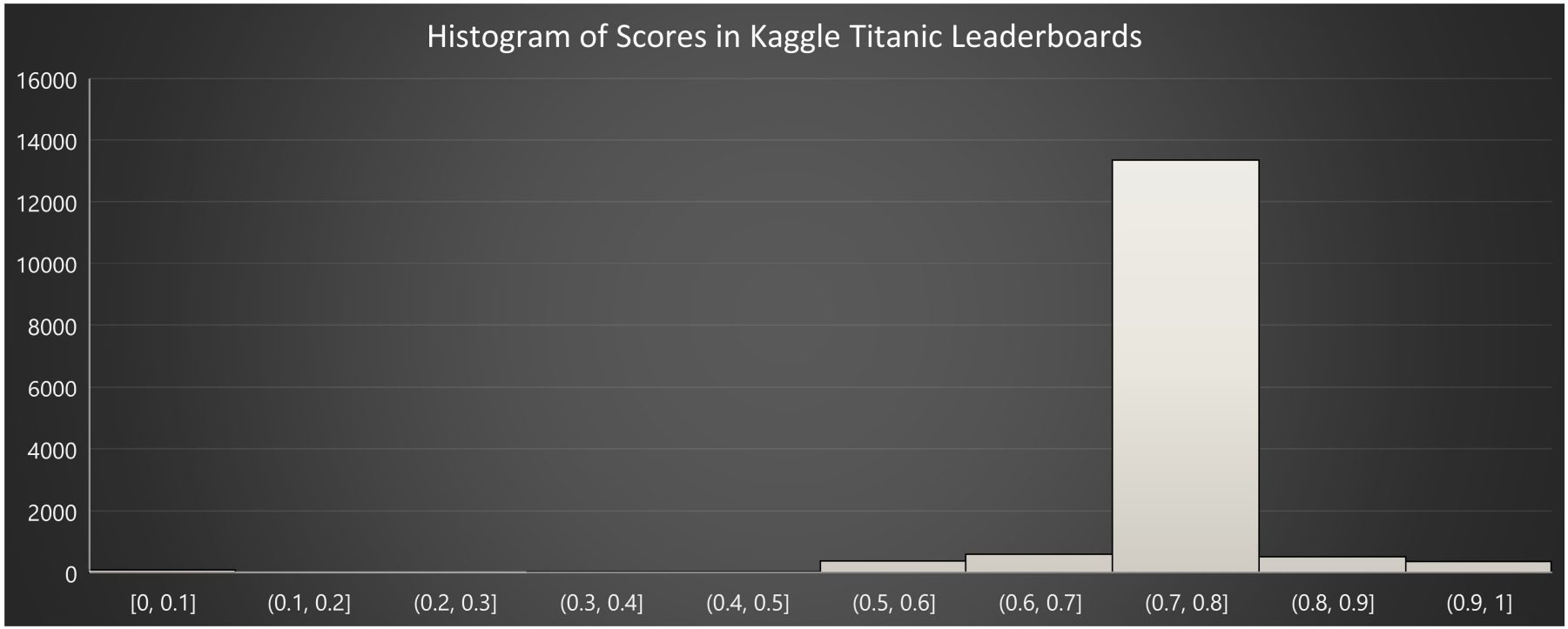
```
==2901== NVPROF is profiling process 2901, command: ./data 0 1000000
==2901== Profiling application: ./data 0 1000000
==2901== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   64.53%  28.100ms         1  28.100ms  28.100ms  28.100ms  init(curandStateXORWOW*, unsigned long, int)
                   28.56%  12.438ms         8  1.5548ms  1.5491ms  1.5613ms  [CUDA memcpy DtoH]
                    1.88%  817.50us         2  408.75us  386.32us  431.18us  generateLognormal(curandStateXORWOW*, float*, int, float, float, int
                    1.82%  791.61us         2  395.81us  394.67us  396.94us  generateNormal(curandStateXORWOW*, float*, int, float, float, int)
                    1.61%  699.61us         2  349.80us  348.20us  351.41us  generateCategorical2(curandStateXORWOW*, int*, int, float)
                    1.60%  698.71us         2  349.36us  349.29us  349.42us  generateCategorical3(curandStateXORWOW*, int*, int, float, float)
```

# ACCURACY AS N GROWS

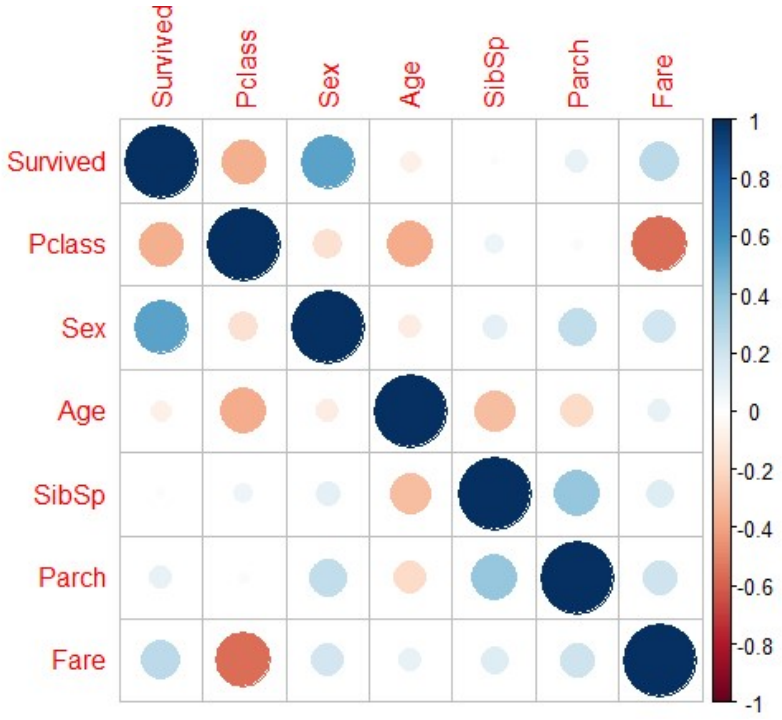


# BASIS FOR A DECENT MODEL



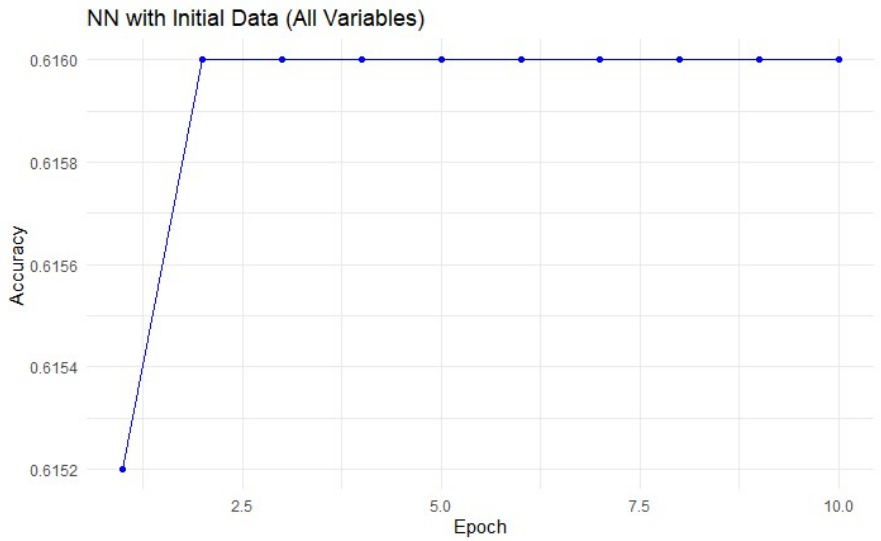
# VARIABLE SELECTION




Variables Selected to use for more tuned modeling
- PClass
- Sex
- Fare

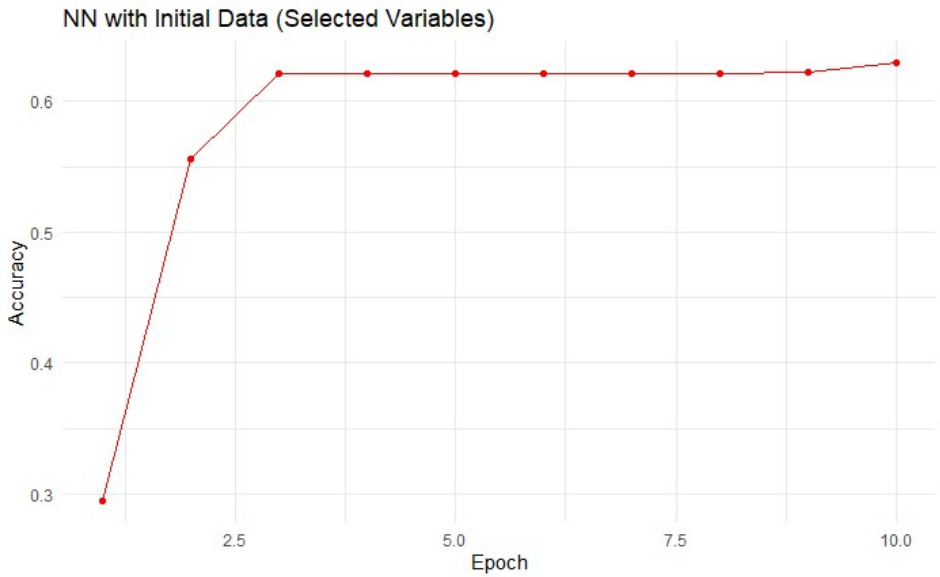
# MODELING NORMAL RESULTS

- **Logistic Regression & Random Forest:** High accuracy (0.81) with better precision for non-survivors.
- **TensorFlow Neural Network:** Reaches 0.8045 accuracy, comparable to other models in Titanic dataset classification.



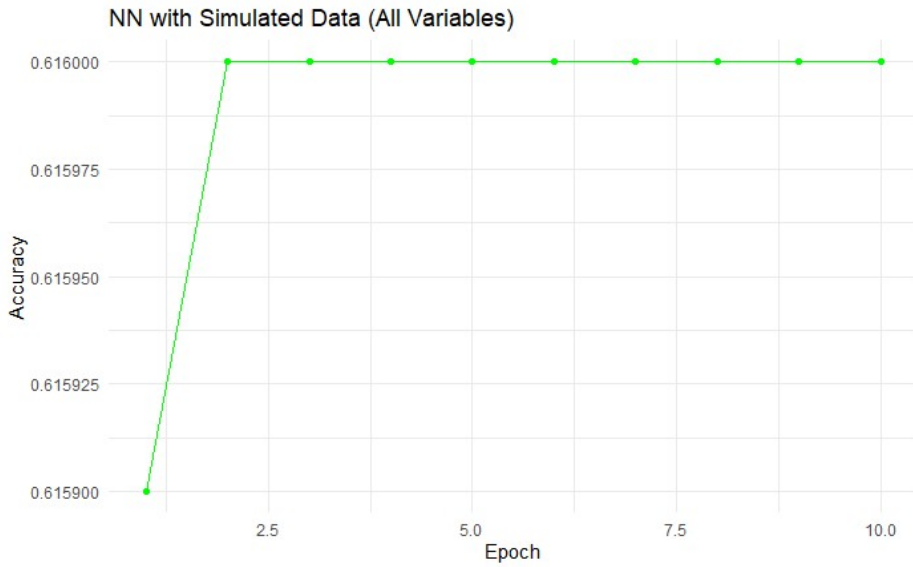
# MODELING NORMAL RESULTS WITH VARIABLE SELECTION

- **Logistic Regression (Variable Selection):** Balanced, 0.78 accuracy, better at predicting non-survivors.
- **Random Forest (Variable Selection):** Enhanced accuracy at 0.83, excels in non-survivor predictions.
- **TensorFlow Neural Network (Variable Selection):** Learns effectively, reaching about 74.86% accuracy.



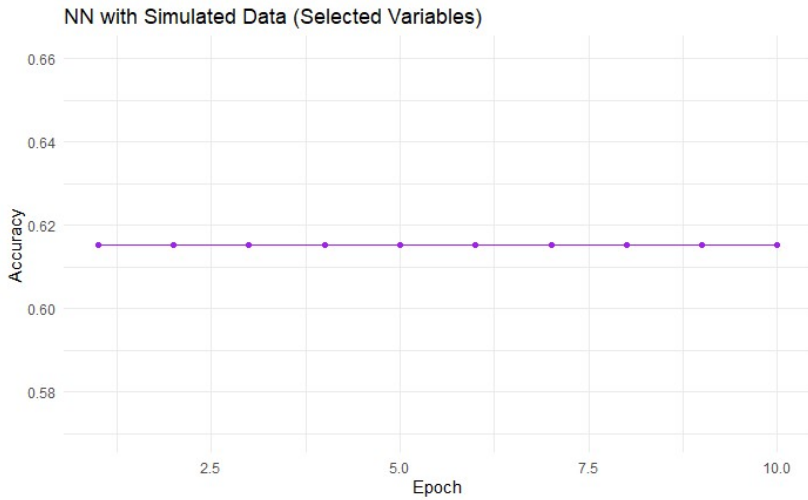
# MODELING SYNTHETIC DETAILS

- **Logistic Regression (Synthetic Data):** Biased towards Survived with 0.62 accuracy; fails to predict Not Survived
- **Random Forest (Synthetic Data):** Moderately accurate (0.54); better at predicting Survived than Not Survived, needs model refinement.
- **TensorFlow Neural Network (Synthetic Data):** Moderately accurate (around 0.615); shows early training convergence, suggesting model or data limitations.



# MODELING SYNTHETIC VARIABLE SELECTION

- **Logistic Regression on Synthetic Data**: High recall for Survived, fails for Not Survived, 62% accuracy, showing bias towards Class 1.
- **Random Forest on Synthetic Data**: Moderate recall for Survived, low for Not Survived, 58% accuracy, slightly more balanced than Logistic Regression.
- **TensorFlow Neural Network on Synthetic Data**: Steady 61.6% accuracy, suggesting moderate performance and possible model/data limitations.



# THE SINKING OF MY MONTE CARLO

- **Time Constraints and Methodology:** Faced with limited time, extensive exploration of various coding methods was prioritized over the application of Markov chain for introducing probabilistic parameters in random data generation. Wanted to start some form of  Cholesky Decomposition but ran out of time
- **Study on Randomness:** This research offers intriguing insights into the performance of models under conditions where individuals in the dataset are assigned variables randomly, without any inter-variable correlation.
- **Simulated Data for Academia and Research:** The project underscores the potential of using simulated data in academic and research contexts, particularly for studies focusing on model performance under varying data conditions.

•**Experiment with Data Modeling:** Utilize the **data modeling**, **fun**, and **fun_gpu** modules, which offer interactive, Oregon Trail-style experiences in data creation and manipulation.
•**Demonstration:** A walkthrough will be provided to showcase the functionality and features of these modules.

# TIME FOR FUN

- Clone the Repository: Access and clone the provided repository to explore its contents.
- Experiment with Data Modeling: Utilize the data modeling, fun, and fun_gpu modules, which offer interactive, Oregon Trail-style experiences in data creation and manipulation.
- Demonstration: A walkthrough will be provided to showcase the functionality and features of these modules.
- NOTE(Maximum safe N for simultaneous operations approximately: 3,66,090,240)

# REFERENCE


- https://drive.google.com/file/d/1VmkAAGOYCTORq1wxSQqy255qLJjTNvBI/edit (374)
- https://medium.com/@rahul.singh.ds20/synthetic-data-future-of-data-6efd0b06a8ad
- https://yingqijing.medium.com/multivariate-normal-distribution-and-cholesky-decomposition-in-stan-d9244b9aa623
- https://www.tensorflow.org/guide/keras/sequential_model
- https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense
- https://www.sciencedirect.com/topics/mathematics/lognormal-distribution
- https://www.kaggle.com/competitions/titanic
- https://www.anesi.com/titanic.htm