Homework 9: ISYE 6501 - Introduction to Analytics Modeling

Question 12.1

Prompt

Describe a situation or problem from your job, everyday life, current events, etc., for which a design of experiments approach would be appropriate.

Solution

A Design of Experiments (DOE) approach would be useful in studying how different factors influence a team's offensive performance. We could design an experiment to analyze the impact of various play-calling strategies on the success rate of scoring drives.

Effect of Play-Calling Strategy on Scoring Success

NFL teams are constantly trying to optimize their offensive strategy. They may want to understand how different combinations of pass, run, and trick plays affect their likelihood of scoring on a drive. Teams need to balance their play selection against different defensive schemes, field positions, and in-game situations (down and distance, clock management, etc.).

Factors (Independent Variables):

- Ratio of passing to rushing plays (e.g., 60% pass / 40% run, 50% pass / 50% run)
- Type of passing plays (short, medium, long, play-action)
- Run types (inside, outside, zone, power)
- Play tempo (fast-paced, slow-paced)
- Defensive scheme (e.g., 4-3, 3-4, nickel, zone coverage)

Response Variable (Dependent Variable):

- Drive success rate (e.g., touchdown, field goal, punt)

Question 12.2

Prompt

To determine the value of 10 different yes/no features to the market value of a house (large yard, solar roof, etc.), a real estate agent plans to survey 50 potential buyers, showing a fictitious house with different combinations of features.
To reduce the survey size, the agent wants to show just 16 fictitious houses. Use R's `FrF2` function (in the `FrF2` package) to find a fractional factorial design for this experiment: what set of features should each of the 16 fictitious houses have?
Note: the output of `FrF2` is "1" (include) or "-1" (don't include) for each feature.

Solution

In a full factorial design with 10 features, each feature has two levels: "1" (include the feature) or "-1" (don't include the feature). With 10 features, the full design would require $2^{10} = 1024$ combinations of features. A fractional factorial design reduces this number by considering only a subset of the combinations while still capturing important information about the main effects and interactions between features.

The `FrF2` function in R helps generate these fractional designs with regular fractional factorial 2-level designs [1]. It takes parameters like the number of factors (features), the number of runs (combinations of features), and other options that control the design's resolution (the ability to distinguish between effects).

Using `FrF2`, 16 rows are generated with 10 columns, each with a 1 to include the feature -1 to not include the feature.

```r
library(FrF2)

# 16 nruns (ficticious houses)
# 10 factors (yes/no features)
frf2_model <- FrF2(nfactors = 10, nruns = 16)
print(frf2_model)
```

```
##     A  B  C  D  E  F  G  H  J  K
## 1   1 -1 -1 -1 -1 -1  1 -1 -1 -1
## 2  -1 -1  1 -1  1 -1 -1  1  1 -1
## 3  -1 -1 -1  1  1  1  1 -1  1 -1
## 4   1 -1 -1  1 -1 -1  1  1  1  1
## 5   1 -1  1 -1 -1  1 -1 -1  1  1
## 6  -1 -1  1  1  1 -1 -1 -1 -1  1
## 7  -1  1  1  1 -1 -1  1 -1  1 -1
## 8   1 -1  1  1 -1  1 -1  1 -1 -1
## 9  -1  1 -1  1 -1  1 -1 -1 -1  1
## 10  1  1  1  1  1  1  1  1  1  1
## 11 -1 -1 -1 -1  1  1  1  1 -1  1
## 12 -1  1  1 -1 -1 -1  1  1 -1  1
## 13  1  1 -1 -1  1 -1 -1 -1  1  1
## 14 -1  1 -1 -1 -1  1 -1  1  1 -1
## 15  1  1  1 -1  1  1  1 -1 -1 -1
## 16  1  1 -1  1  1 -1 -1  1 -1 -1
## class=design, type= FrF2
```

Question 13.1

Prompt

For each of the following distributions, give an example of data that you would expect to follow this distribution (besides the examples already discussed in class). a. Binomial
b. Geometric
c. Poisson
d. Exponential
e. Weibull

Solution

Considering each distribution in american football or the National Football League (NFL) examples are provided to model different types of NFL-related events.

Binomial Distribution

The number of successful pass completions out of a certain number of attempts in a game. Each pass attempt can either be a success (completion) or failure (incompletion), with a fixed probability of success.

Geometric Distribution

The number of plays until a touchdown is scored. This assumes that each play has a constant probability of resulting in a touchdown, and you're counting the number of trials until the first success (touchdown).

Poisson Distribution

The number of touchdowns scored by a team in a game. This counts the number of events (touchdowns) that occur in a fixed interval of time (the game), where the events happen independently.

Exponential Distribution

The time between two consecutive scoring plays in a game (touchdowns or field goals). The exponential distribution models the time between independent events occurring at a constant rate.

Weibull Distribution

The time until a player gets injured. The Weibull distribution is often used to model failure times or life durations, which could relate to how long a player plays before getting injured. Different shapes of the Weibull distribution can account for "wear and tear" effects that might increase or decrease risk over time.

Question 13.2

Prompt

In this problem you, can simulate a simplified airport security system at a busy airport. Passengers arrive according to a Poisson distribution with $\lambda_1 = 5$ per minute (i.e., mean interarrival rate $\mu_1 = 0.2$ minutes) to the ID/boarding-pass check queue, where there are several servers who each have exponential service time with mean rate $\mu_2 = 0.75$ minutes. [Hint: model them as one block that has more than one resource.] After that, the passengers are assigned to the shortest of the several personal-check queues, where they go through the personal scanner (time is uniformly distributed between 0.5 minutes and 1 minute).

Use the Arena software (PC users) or Python with SimPy (PC or Mac users) to build a simulation of the system, and then vary the number of ID/boarding-pass checkers and personal-check queues to determine how many are needed to keep average wait times below 15 minutes. [If you're using SimPy, or if you have access to a non-student version of Arena, you can use $\lambda_1 = 50$ to simulate a busier airport.]

Solution

SimPy is a process-based discrete event simulation framework [2]. Using SimPy with Python, the airport security system can be modeled with a few classes that are created to execure the processes with multiple functions. **There are 3 classes and a lot of code.**

`SecuritySystem` represents the security system with boarding pass checkers and scanners. The class uses attributes, `env`, `server`, and `scanner`. `env` is the SimPy environment. `server` is a SimPy resource representing boarding pass checkers. `scanner` A list of SimPy resources representing scanners. The `scanner` is processed in a for loop such tha each scanner is represented in a queue. `checkBoardingPass(serviceTime)` is a function that simulates the time taken to check a boarding pass using an exponential distribution. `scanPerson(minScan, maxScan)` is a function that simulates the time taken to scan a person using a uniform distribution.

The second class, `Passenger`, represents a passenger going through the system. The main function is `run` is the process that simulates a passenger's journey through the security system. It includes arriving at the checkpoint, requesting and waiting for a boarding pass checker, requesting and waiting for a scanner, and recording the times taken for each step and the total time in the system.

The `Simulation` class controls everything, managing the overall simulation, including setup, running multiple replications, and plotting results. `setup(env)` sets up the simulation environment, including initializing the security system and generating passengers. `run()` runs the simulation for the specified number of replications and for each replication it initializes the SimPy environment. The function then runs the simulation for the specified duration. Finally, it collects and prints average times for each replication. Two functions are created for visualizing the data. `plot_results()` plots the average times for checking, scanning, waiting, and total system time across replications. `plot_histogram()` plots a histogram of the total system times.

Finally, the remaining code defines the parameters for the simulation, including the number of checkers, scanners, arrival rate, service time, scan times, run time, and replications. Creates an instance of the Simulation class with the specified parameters. The simulation is then ran and we will plot the results after.

A 2 hour time period seems reasonable, so a `runTime` of 720 minutes is used. We run a distribution of 100 replications to provide a good sample size for an estimate.

```
import random
import simpy
import matplotlib.pyplot as plt


# Class representing the security system
```

```
class SecuritySystem:
    def __init__(self, env, num_scanners, num_servers):
        self.env = env
        self.server = simpy.Resource(env, num_servers)  # Boarding pass checkers
        self.scanner = [simpy.Resource(env, 1) for _ in range(num_scanners)]  # Scanners

    # Process for checking boarding pass
    def checkBoardingPass(self, serviceTime):
        yield self.env.timeout(random.expovariate(1.0 / serviceTime))

    # Process for scanning a person
    def scanPerson(self, minScan, maxScan):
        yield self.env.timeout(random.uniform(minScan, maxScan))

# Class representing a passenger
class Passenger:
    def __init__(self, env, name, system, simulation, serviceTime, minScan, maxScan):
        self.env = env
        self.name = name
        self.system = system
        self.simulation = simulation
        self.serviceTime = serviceTime
        self.minScan = minScan
        self.maxScan = maxScan
        self.action = env.process(self.run())  # Start the passenger process

    # Process for a passenger going through the system
    def run(self):
        timeArrive = self.env.now

        # Request to check boarding pass
        with self.system.server.request() as request:
            yield request
            tIn = self.env.now
            yield self.env.process(self.system.checkBoardingPass(self.serviceTime))
            tOut = self.env.now
            self.simulation.checkTime.append(tOut - tIn)

        # Find the scanner with the shortest queue
        minq = min(
            range(len(self.system.scanner)),
            key=lambda i: len(self.system.scanner[i].queue),
        )

        # Request to scan
        with self.system.scanner[minq].request() as request:
            yield request
            tIn = self.env.now
            yield self.env.process(self.system.scanPerson(self.minScan, self.maxScan))
            tOut = self.env.now
            self.simulation.scanTime.append(tOut - tIn)

        timeLeave = self.env.now
```

```
        self.simulation.sysTime.append(timeLeave - timeArrive)
        self.simulation.totThrough += 1

# Class representing the simulation
class Simulation:
    def __init__(
        self,
        numCheckers,
        numScanners,
        arrivalRate,
        serviceTime,
        minScan,
        maxScan,
        runTime,
        replications,
    ):
        self.numCheckers = numCheckers
        self.numScanners = numScanners
        self.arrivalRate = arrivalRate
        self.serviceTime = serviceTime
        self.minScan = minScan
        self.maxScan = maxScan
        self.runTime = runTime
        self.replications = replications
        self.avgCheckTime = []
        self.avgScanTime = []
        self.avgWaitTime = []
        self.avgSystemTime = []

    # Setup the simulation environment
    def setup(self, env):
        self.totThrough = 0
        self.checkTime = []
        self.scanTime = []
        self.sysTime = []
        system = SecuritySystem(env, self.numScanners, self.numCheckers)
        i = 0
        while True:
            yield env.timeout(random.expovariate(self.arrivalRate))
            i += 1
            Passenger(
                env,
                f"Passenger {i}",
                system,
                self,
                self.serviceTime,
                self.minScan,
                self.maxScan,
            )

    # Run the simulation
    def run(self):
        for i in range(self.replications):
```

```
            random.seed(i)
            env = simpy.Environment()
            env.process(self.setup(env))
            env.run(until=self.runTime)

            self.avgSystemTime.append(sum(self.sysTime) / self.totThrough)
            self.avgCheckTime.append(sum(self.checkTime) / self.totThrough)
            self.avgScanTime.append(sum(self.scanTime) / self.totThrough)
            self.avgWaitTime.append(
                self.avgSystemTime[i] - self.avgCheckTime[i] - self.avgScanTime[i]
            )

            print(
                f"{self.totThrough} : Replication {i+1} times {self.avgSystemTime[i]:.2f} {self.avgChec
            )

        print("-----")
        print(
            f"Average system time = {sum(self.avgSystemTime) / self.replications:.2f}"
        )
        print(f"Average check time = {sum(self.avgCheckTime) / self.replications:.2f}")
        print(f"Average scan time = {sum(self.avgScanTime) / self.replications:.2f}")
        print(f"Average wait time = {sum(self.avgWaitTime) / self.replications:.2f}")

    # Plot the results of the simulation
    def plot_results(self):
        plt.figure(figsize=(12, 8))

        # Top left plot
        plt.subplot(2, 2, 1)
        plt.plot(self.avgSystemTime, label="Avg. System Time")
        plt.xlabel("Replication")
        plt.ylabel("Time (minutes)")
        plt.title("Average System Time")
        plt.legend()
        plt.grid()

        # Top right plot
        plt.subplot(2, 2, 2)
        plt.plot(self.avgCheckTime, label="Avg. Check Time")
        plt.xlabel("Replication")
        plt.ylabel("Time (minutes)")
        plt.title("Average Check Time")
        plt.legend()
        plt.grid()

        # Bottom left plot
        plt.subplot(2, 2, 3)
        plt.plot(self.avgScanTime, label="Avg. Scan Time")
        plt.xlabel("Replication")
        plt.ylabel("Time (minutes)")
        plt.title("Average Scan Time")
        plt.legend()
```

```
        plt.grid()

        # Bottom right plot
        plt.subplot(2, 2, 4)
        plt.plot(self.avgWaitTime, label="Avg. Wait Time")
        plt.xlabel("Replication")
        plt.ylabel("Time (minutes)")
        plt.title("Average Wait Time")
        plt.legend()
        plt.grid()

        plt.tight_layout()
        plt.show()

    # Plot histogram of system times
    def plot_histogram(self):
        plt.figure(figsize=(12, 8))

        plt.hist(self.sysTime, bins=30, alpha=0.7, label="System Time")
        plt.xlabel("Time (minutes)")
        plt.ylabel("Frequency")
        plt.title("Histogram of System Times")
        plt.legend()

        plt.show()

# ID/Boarding Pass Checkers
numCheckers = 37
# Scanners
numScanners = 37
arrivalRate = 50
serviceTime = 0.75
# Uniform distribution for scanner time
minScan = 0.5  # Minimum time
maxScan = 1.0  # Maximum time
# Run time and iterations
runTime = 720  # Run for X minutes
replications = 100  # Number of runs

sim = Simulation(
    numCheckers,
    numScanners,
    arrivalRate,
    serviceTime,
    minScan,
    maxScan,
    runTime,
    replications,
)
sim.run()
```

```
## 35396 : Replication 1 times 5.73 0.75 0.75 4.23
## 35172 : Replication 2 times 9.56 0.75 0.75 8.05
```

```
## 35301 : Replication 3 times 9.34 0.75 0.75 7.84
## 35298 : Replication 4 times 10.46 0.75 0.75 8.96
## 35255 : Replication 5 times 6.46 0.75 0.75 4.96
## 35224 : Replication 6 times 11.45 0.75 0.75 9.94
## 35181 : Replication 7 times 6.61 0.75 0.75 5.11
## 35276 : Replication 8 times 6.77 0.75 0.75 5.27
## 35390 : Replication 9 times 9.09 0.75 0.75 7.59
## 35185 : Replication 10 times 12.11 0.75 0.75 10.61
## 35358 : Replication 11 times 9.06 0.75 0.75 7.56
## 35396 : Replication 12 times 7.12 0.75 0.75 5.62
## 35131 : Replication 13 times 11.25 0.75 0.75 9.75
## 35386 : Replication 14 times 5.84 0.75 0.75 4.34
## 35183 : Replication 15 times 13.99 0.75 0.75 12.49
## 35158 : Replication 16 times 13.03 0.76 0.75 11.52
## 35127 : Replication 17 times 10.74 0.76 0.75 9.23
## 35364 : Replication 18 times 9.15 0.75 0.75 7.65
## 35355 : Replication 19 times 9.04 0.75 0.75 7.54
## 35359 : Replication 20 times 9.27 0.75 0.75 7.77
## 35161 : Replication 21 times 7.27 0.76 0.75 5.76
## 35247 : Replication 22 times 9.81 0.75 0.75 8.31
## 35095 : Replication 23 times 10.55 0.76 0.75 9.05
## 35088 : Replication 24 times 7.63 0.76 0.75 6.13
## 35390 : Replication 25 times 10.31 0.75 0.75 8.81
## 35154 : Replication 26 times 10.67 0.76 0.75 9.16
## 35328 : Replication 27 times 10.80 0.75 0.75 9.29
## 35089 : Replication 28 times 9.93 0.76 0.75 8.42
## 35259 : Replication 29 times 10.17 0.75 0.75 8.67
## 35265 : Replication 30 times 10.43 0.75 0.75 8.92
## 35373 : Replication 31 times 11.13 0.75 0.75 9.63
## 34887 : Replication 32 times 8.41 0.76 0.75 6.90
## 35195 : Replication 33 times 11.86 0.75 0.75 10.36
## 35215 : Replication 34 times 7.44 0.76 0.75 5.94
## 35292 : Replication 35 times 8.96 0.75 0.75 7.46
## 35389 : Replication 36 times 8.45 0.75 0.75 6.95
## 35304 : Replication 37 times 7.95 0.75 0.75 6.45
## 35249 : Replication 38 times 9.11 0.75 0.75 7.61
## 35196 : Replication 39 times 11.54 0.76 0.75 10.04
## 35104 : Replication 40 times 12.30 0.76 0.75 10.80
## 35389 : Replication 41 times 7.30 0.75 0.75 5.80
## 35280 : Replication 42 times 6.31 0.75 0.75 4.81
## 35089 : Replication 43 times 13.78 0.76 0.75 12.28
## 35152 : Replication 44 times 8.47 0.75 0.75 6.96
## 34992 : Replication 45 times 5.86 0.76 0.75 4.35
## 35261 : Replication 46 times 11.26 0.75 0.75 9.76
## 35249 : Replication 47 times 6.68 0.75 0.75 5.18
## 35004 : Replication 48 times 11.24 0.76 0.75 9.73
## 35152 : Replication 49 times 9.51 0.76 0.75 8.01
## 35295 : Replication 50 times 8.75 0.75 0.75 7.25
## 35065 : Replication 51 times 7.39 0.76 0.75 5.89
## 35209 : Replication 52 times 9.09 0.76 0.75 7.58
## 35225 : Replication 53 times 7.85 0.75 0.75 6.34
## 35119 : Replication 54 times 10.62 0.76 0.75 9.11
## 35309 : Replication 55 times 10.46 0.75 0.75 8.95
## 35231 : Replication 56 times 9.25 0.75 0.75 7.75
```

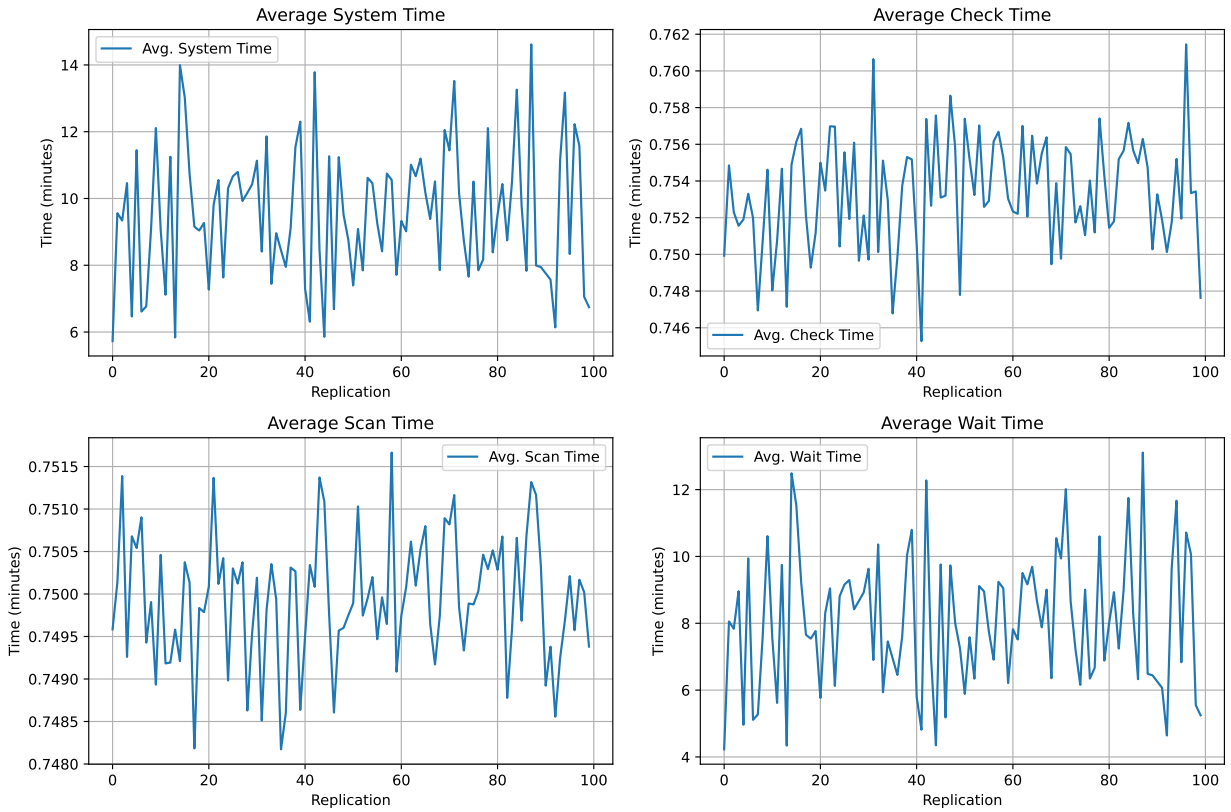
```
## 35023 : Replication 57 times 8.42 0.76 0.75 6.91
## 35062 : Replication 58 times 10.75 0.76 0.75 9.24
## 35184 : Replication 59 times 10.56 0.76 0.75 9.05
## 35327 : Replication 60 times 7.71 0.75 0.75 6.21
## 35304 : Replication 61 times 9.32 0.75 0.75 7.82
## 35348 : Replication 62 times 9.02 0.75 0.75 7.51
## 35079 : Replication 63 times 11.01 0.76 0.75 9.51
## 35323 : Replication 64 times 10.67 0.75 0.75 9.16
## 35132 : Replication 65 times 11.20 0.76 0.75 9.69
## 35281 : Replication 66 times 10.17 0.75 0.75 8.67
## 35125 : Replication 67 times 9.38 0.76 0.75 7.88
## 35159 : Replication 68 times 10.51 0.76 0.75 9.01
## 35291 : Replication 69 times 7.85 0.75 0.75 6.36
## 35239 : Replication 70 times 12.05 0.75 0.75 10.55
## 35333 : Replication 71 times 11.44 0.75 0.75 9.94
## 35187 : Replication 72 times 13.52 0.76 0.75 12.01
## 35172 : Replication 73 times 10.16 0.76 0.75 8.65
## 35374 : Replication 74 times 8.72 0.75 0.75 7.22
## 35307 : Replication 75 times 7.66 0.75 0.75 6.16
## 35357 : Replication 76 times 10.51 0.75 0.75 9.01
## 35154 : Replication 77 times 7.85 0.75 0.75 6.35
## 35338 : Replication 78 times 8.17 0.75 0.75 6.67
## 35093 : Replication 79 times 12.11 0.76 0.75 10.60
## 35160 : Replication 80 times 8.39 0.75 0.75 6.88
## 35307 : Replication 81 times 9.52 0.75 0.75 8.01
## 35327 : Replication 82 times 10.43 0.75 0.75 8.93
## 35168 : Replication 83 times 8.75 0.76 0.75 7.24
## 35163 : Replication 84 times 10.50 0.76 0.75 8.99
## 35084 : Replication 85 times 13.26 0.76 0.75 11.75
## 35112 : Replication 86 times 9.77 0.76 0.75 8.27
## 35214 : Replication 87 times 7.83 0.75 0.75 6.33
## 35150 : Replication 88 times 14.62 0.76 0.75 13.11
## 35186 : Replication 89 times 7.99 0.75 0.75 6.49
## 35286 : Replication 90 times 7.95 0.75 0.75 6.44
## 35180 : Replication 91 times 7.76 0.75 0.75 6.26
## 35342 : Replication 92 times 7.57 0.75 0.75 6.07
## 35373 : Replication 93 times 6.14 0.75 0.75 4.64
## 35366 : Replication 94 times 11.15 0.75 0.75 9.65
## 35156 : Replication 95 times 13.17 0.76 0.75 11.67
## 35326 : Replication 96 times 8.34 0.75 0.75 6.83
## 34829 : Replication 97 times 12.23 0.76 0.75 10.72
## 35297 : Replication 98 times 11.57 0.75 0.75 10.07
## 35271 : Replication 99 times 7.05 0.75 0.75 5.55
## 35380 : Replication 100 times 6.74 0.75 0.75 5.25
## -----
## Average system time = 9.54
## Average check time = 0.75
## Average scan time = 0.75
## Average wait time = 8.04
```

For the busy airport where arrival rate is equal to 50, the optimal choice is 37 ID/Boarding Pass Checkers and 37 Scanners to keep the average wait time below 15 minutes. The wait time is about 7 minutes, below the target of 15 minutes. Using 36 for both would result in a 17 minute wait time.

Plotting the results, we can see the average system time, check time, scan time, and wait time across

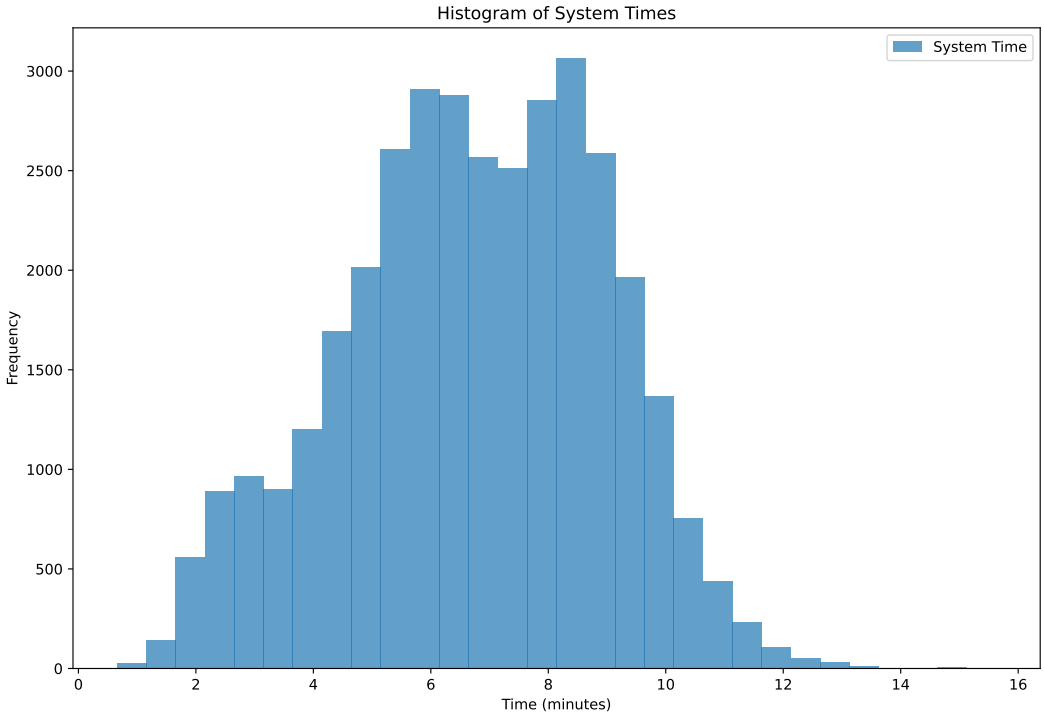
replications. Each scanner/checker is almost equal in time, so there is not a significant difference between the two.

```
sim.plot_results()
```



The histogram shows the distribution of system times. Most passengers go through the system in about 7 minutes, but there are some outliers that take longer.

```
sim.plot_histogram()
```



Smaller/Less Busy Airports:

Smaller or less busy airports with an arrival rate of 5 is optimal with 4 ID/Boarding Pass Checkers and 4 Scanners to keep the average wait time below 15 minutes. The wait time is about 4 minutes, below the target of 15 minutes. If the number of checkers/scanners is reduced to 3, the wait time increases to around 75 minutes!

References

[1] "FrF2: Function to provide regular fractional factorial 2-level designs - r documentation — rdocumentation.org." https://www.rdocumentation.org/packages/FrF2/versions/2.3-3/topics/FrF2.

[2] "SimPy - discrete event simulation in python." https://simpy.readthedocs.io/en/latest/index.html.