RE-EVALUATING PERFORMANCE MEASUREMENT: NEW MATHEMATICAL METHODS TO ADDRESS COMMON PERFORMANCE MEASUREMENT CHALLENGES

A Thesis

Submitted to the McAnulty Graduate School of Liberal Arts

Duquesne University

In partial fulfillment of the requirements for

the degree of Masters of Science in Computational Mathematics

By

Jordan Benis

May 2018

RE-EVALUATING PERFORMANCE MEASUREMENT: NEW

MATHEMATICAL METHODS TO ADDRESS COMMON PERFORMANCE

MEASUREMENT CHALLENGES

By

Jordan Benis

Approved April 10, 2018

_____

Dr. John C. Kern II, Department Chair
Associate Professor of Statistics
(Committee Chair)

_____

Dr. Frank D'Amico
Professor of Statistics
(Committee Member)

_____

Dr. Jim Swindal, Dean
McAnulty College of Graduate School of Liberal Arts

ABSTRACT

RE-EVALUATING PERFORMANCE MEASUREMENT: NEW MATHEMATICAL METHODS TO ADDRESS COMMON PERFORMANCE MEASUREMENT CHALLENGES

By

Jordan. D. Benis

May 2018

Thesis supervised by Dr. John C. Kern II

Performance Measurement is an essential discipline for any business. Robust and reliable performance metrics for people, processes, and technologies enable a business to identify and address deficiencies to improve performance and profitability. The complexity of modern operating environments presents real challenges to developing equitable and accurate performance metrics. This thesis explores and develops two new methods to address common challenges encountered in businesses across the world. The first method addresses the challenge of estimating the relative complexity of various tasks by utilizing the Pearson Correlation Coefficient to identify potentially over weighted and under weighted tasks. The second method addresses the challenge of determining performers' influence on a metric by treating performance rankings as vectors and evaluating the change of the vector over multiple performance periods.

ACKNOWLEDGMENT

I would like to thank Dr. John Kern for his guidance, insights, and support throughout the research process. I would also like to thank Dr. Kern and the rest of the CPMA faculty at Duquesne University for their tireless dedication to serving students through teaching in the fields of Mathematics, Statistics, and Computer Science. Their guidance and teaching have enabled me to develop into a more effective student, professional, and thinker. Lastly, I would like to thank my wife for her continued emotional support throughout my graduate studies.

TABLE OF CONTENTS

Introduction

Performance Measurement is a critical discipline for any business. Being able to consistently measure the performance of people, processes, and technology enables businesses to identify opportunities for improvement and evolve to adapt to ever-changing market and industry conditions. Despite its importance, Performance Measurement offers many challenges, especially with respect to the performance of people.

Performance Measurement is a well-studied field. Mathematicians, psychologists, and industrial engineers have produced volumes of work on observations, methods, and insights related to Performance Measurement. For example, Dr. James B. Schreiber (2016) has published literature on motivation and how it affects performance. Schreiber explores how factors such as incentives, collaboration, and rewards impact motivation and ultimately performance. Kathryn E. Merrick and Shafi Kamran (2011) have developed closely related ideas into a robust mathematical model that defines the relationship between various motivational factors and performance. Many other works have been published to approach productivity and worker behavior from a psychological and mathematical perspective. Many of these studies assume that productivity and performance can be reliably measured. This assumption, while appropriate in many cases, presupposes that many challenges can and have been addressed. Exploring these challenges and methods to address those challenges will be the primary focus of this paper.

Performance Measurement Overview

Strong people performance metrics must meet several criteria to ensure their validity and accuracy. Most importantly, these metrics must be critical to the priorities of the business, able to be reliably and accurately measured, and directly impacted by the performance of the individuals. This last two criteria are often the most difficult to assess. Determining whether a metric relates to critical business priorities involves little more than evaluating how the outcome of the metric relates to the welfare of one of a business's key stakeholders, which are the shareholder, the employee, the customer, and the community. However, designing a metric that reliably and accurately measures performance is challenging due to many complicating factors that will be discussed in the next section. Additionally, determining whether a metric can be sufficiently impacted by each individual is also challenging, as many performance metrics are impacted by multiple factors that are beyond the control of the individual performer.

I. Capacity Weighting Correlation Analysis

Analysis Background

One of the most common complicating factors that can impact the ability to reliably measure a performance arises from the variation in the amount of time and effort required to complete different types of tasks. Although some production roles involve performing one single task repetitively, many modern jobs require an employee to perform multiple different tasks with varying frequencies. The fact that each task may vary in complexity makes consistently measuring productivity or planning for capacity challenging. To accurately measure

productivity across different performers, a system must account for the fact that different tasks take a different amount of time to complete. To properly plan for capacity, a model must account for the differing amounts of available capacity that each task will absorb. Understanding the relative amount of time it takes to perform each task is, therefore, critical to effective operations management.

Common Approaches to Capacity Weighting Problem

A common approach to determine the relative amount of time required to complete different types of tasks is to perform a time study. Such a study involves timing a sample of the tasks being performed and using the results to estimate how long each task takes. This common approach presents an array of problems. First, it is often prohibitively expensive or challenging to gather a sample size that is large enough to yield a reliable estimate. This is especially true for tasks that vary greatly in the amount of time to complete. Even if a large enough sample can be gathered, a common phenomenon known as the "Hawthorne Effect" could skew the results. (Machol 1975) This effect describes the human tendency to perform differently when being observed. The term for the effect was coined after experiments were performed at the Hawthorn plant near Chicago in the 1930's to determine the environment's effect on worker productivity. The study reportedly revealed that the act of observing the productivity of the workers itself impacted how productive the workers were. Although the apocryphal origin of the discovery of the effect has since been called into question, the existence of the Hawthorne effect is still widely accepted and limits the usefulness of time studies. (Machol 1975) For example, time study participants may complete tasks more quickly when they know they are being timed.

Other common sampling challenges, such as selection bias, can also skew time study results. If workflow is entirely automated, the workflow system will likely be able to capture timestamps at the beginning and end of each task which could eliminate these sampling challenges. Many operating environments, however, do not use a fully automated workflow system. Additionally, the automated timings may not account for follow-up work that may occur more frequently for certain tasks.

Capacity Weighting Correlation Analysis – Unadjusted for Productivity

Due to the importance of understanding relative timings and the difficulty of obtaining them, a method that reliably estimates relative timings using historical data would be very valuable in any operating environment where performers are completing multiple tasks. This analysis will use terms that are defined below in Table 1:

Term	**Definition**
Task	A process or function performed by a performer
Task Completion Time	The average amount of time it takes to complete all of the responsibilities associated with a task
Actual Task Weight	A weight that corresponds to the relative Task Completion Time
Assigned Task Weight	The weight that is assigned to a task to calculate a weighted productivity score
Under-Weighted Task	A task with an Assigned Task Weight that is **lower** than the Actual Task Weight
Over-Weighted Task	A task with an Assigned Task Weight that is **higher** than the Actual Task Weight

Table 1

To develop such a method, we will first consider a simple case where a group performs three tasks. The actual task weight will be directly proportional to the task completion time. For example, a task with weight 2 would take twice as much time to perform as a task with weight 1. We will begin by assuming each task is equally weighted by assigning all three tasks an assigned task weight of 1. In this simple case, we will also assume that all performers are equally productive.

Since we have assigned equal weights to all three tasks, tasks that are more complex and take more time are "under-weighted" because the performer is receiving no more credit for these tasks than he would receive for an easier task. Similarly, tasks that are less complex and take less time to complete are "over-weighted" because the performer receives as much credit as he would if he had completed a more complex task. If the equal weighting system is used to calculate productivity scores, the performers who perform more of the under-weighted tasks would be placed at a disadvantage and those who performed more over-weighted tasks would be placed at an advantage. This fact can be exploited in the data to identify which tasks are over-weighted and under-weighted. We will develop this idea into an analysis called the "Capacity Weighting Correlation Analysis".

The Capacity Weighting Correlation Analysis will begin by calculating a productivity score for each performer based on the performance data. Productivity scores will be calculated by taking a sum product of the tasks completed and their corresponding weights. When all assigned weights equal 1, this method is equivalent to a simple sum of the tasks completed. If some tasks take longer to complete than others, one would expect that those performers who were assigned more under-weighted tasks will be put at a disadvantage since they receive no more credit for these under weighted and more complex tasks. That is, completing more under-

weighted tasks will lead to a lower weighted productivity score. Conversely, one would expect that completing more over-weighted tasks would lead to a higher weighted productivity score. The relationship between the number of each task completed and the overall weighted productivity score could be measured using several common methods; such as the Pearson, Kendall, and Spearman correlation methods. Evaluating these methods is beyond the scope of this research. We will therefore use the oldest and most conventional method: The Pearson Correlation Coefficient. (Nicewander 1988)

The Pearson Correlation Coefficient was developed by Karl Pearson in 1895 and is commonly represented by *r*. This coefficient is calculated by dividing the covariance of two paired sets of data by the product of the standard deviations of each set of data. The covariance provides an indication of how closely related two data sets are, with a positive covariance suggesting a positive relationship and a negative covariance suggesting a negative relationship. Dividing the product by the standard deviations yields a coefficient that must lie between -1 and 1 inclusive. A value of -1 suggests a perfect negative correlation while a value of 1 suggests a perfect positive correlation and a value of 0 suggests no correlation at all. The full formula for the Person correlation coefficient is expressed below in (1): (Nicewander 1988)

$$r = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2 \sum_{i=1}^{n}(y_i-\bar{y})^2}}. \quad (1)$$

A simulation will be used to test whether over-weighted tasks consistently yield a larger *r* coefficient and under-weighted tasks consistently yield a smaller *r* coefficient with the overall weighted productivity score. To perform this simulation, we must first generate productivity data. We will assign 100 simulated performers one of three different tasks one at a time. The probability of being assigned each task will be equal. Each of these performers will be assigned

an initial capacity score of 100. When task 1 is completed, 1 capacity point will be deducted. When task 2 is completed, 5 capacity points will be deducted. When task 3 is completed, 10 capacity points will be deducted. That is, the actual task weightings will be 1, 5, and 10. Once a task is assigned that a performer does not have the remaining capacity to complete, the performer will be considered finished for the period and the total number of each task completed will be counted to assign a weighted productivity score.

Although the actual task weights will be 1, 5, and 10, the *assigned* task weights will be 1, 1, and 1. Since all three tasks are given the same weight when scores are calculated, task 1 is over-weighted and task 3 is under-weighted. Therefore, according to the hypothesis described above, the number of total tasks completed that were task 1 should have a lower correlation with the overall weighted productivity score than the number of total tasks completed that were task 3 and the overall weighted productivity score. This is because those performers who complete more of task 1 are given equal credit but perform less work. This would give them more time to complete more tasks so we would expect their scores to be higher than those who completed more of task 3. The correlation between the overall weighted productivity score and the number of tasks completed that were task 2 should be between that of task 1 and task 3. The p-value of the *r* coefficient will also be considered in this simulation for the null hypothesis that the population correlation coefficient is equal to zero. Pearson correlation coefficients follow a student-t distribution and *r* coefficients with a large p-value may be false positives. (Nicewander 1988) The simulation will be run 1,000 times. The code for this simulation can be found in Section 2 of the Appendix. The results of this simulation are below in Table 2:

Task Weighting	Average *r* Coefficient (*proportion of total tasks completed to total score*)	5th Percentile of Correlation Coefficient	95th Percentile of Correlation Coefficient	Average p-value
1	0.92	0.89	0.94	<.00001
5	0.47	0.35	0.59	0.0002
10	-0.62	-0.72	-0.53	<00001

Table 2

As Table 2 illustrates, the over-weighted tasks (task 1) consistently yield a higher correlation coefficient and the under-weighted tasks (task 3) consistently yield a lower correlation coefficient with the overall weighted productivity score, as predicted. The p-value for these correlation coefficients is low, suggesting that the coefficients are statistically significant. To ensure that different correlation coefficients do not emerge when the assigned task weight equals the actual task weight, this same simulation will be run on data that is generated using actual task weights of (1,1,1) instead of (1,5,10). The simulation will still use assigned task weights of (1,1,1) so that the actual and assigned task weights are the ssame. The results are below in Table 3:

Task Names	Average *r* Coefficient (*proportion of total tasks completed to total score*)	5th Percentile of Correlation Coefficient	95th Percentile of Correlation Coefficient	Average p-value
Task 1	-0.0006	-0.17	0.16	0.49
Task 2	0.0036	-0.18	0.16	0.48
Task 3	-0.003	-0.16	0.17	0.5

Table 3

As Table 3 illustrates, the *r* coefficients calculated from the number of tasks completed and the overall weighted score do not differ between tasks when the assigned task weights are

the same as the actual task weights. Additionally, none of these *r* coefficients are statistically significant from 0 as illustrated by the high p-values when the tasks are accurately weighted.

We can exploit this relationship to develop the Capacity Weighting Correlation Analysis. This method will utilize past performance data to estimate the actual task weights of the tasks performed. To begin, the method will calculate productivity scores using historical data under the assumption that all tasks are weighted equally. The *r* coefficients and p-values will then be calculated as they were in the simulation above. If the p-value for the *r* coefficient for any given task is less than 0.05 and the *r* coefficient for that task is less than the average *r* coefficient for all tasks with a p-value less than 0.05, then the weight of the task will be increased by 1%. If the p-value for the *r* coefficient for any given task is less than 0.05 and the *r* coefficient for that task is greater than the average *r* coefficient with a p-value less than 0.05 then the weight of that task will be decreased by 1%. The adjusted weights will be used to recalculate the total score and calculate new *r* coefficients and corresponding p-values. This process will be repeated until no p-values for *r* coefficients are below 0.05. Finally, the calculated weights will be divided by the lowest weight that was calculated. This final step will have the effect of indexing the lowest weight to 1 while not disrupting the relative differences between all weightings. Indexing the weights to 1 will make the results easier to interpret and will provide an easy way to identify the lowest weighted task.

This method will be tested using randomly generated data that simulates performers with equal productivity. When the p-values and *r* coefficients are calculated, the assigned weights will be adjusted and the scores recalculated as prescribed above. Data for the first test will be generated using actual task weights of 1, 5, and 10. That is, the first task will absorb 1 capacity point, the second task will absorb 5 capacity points, and the third task will absorb 10 capacity

tasks. The algorithm will then be run on the resulting data in an attempt to recover those weights. The test will be run 100 times with 50 performers in each test. The goal of the test will be to calculate estimated weights that are as close to the actual task weights of 1, 5, and 10 as possible. The code for this analysis and test can be found in Section 2 of the Appendix. The results are expressed below in Table 4:

Task Names	Actual Weighting	Average calculated weighting	5th Percentile of calculated weighting	95th Percentile of calculated weighting	Average Absolute Value of Error
Task 1	1	1	1	1	0%
Task 2	5	4.0	3.1	5.0	20%
Task 3	10	8.0	6.6	9.7	20%

Table 4

The test works well with actual task weights (1, 5, 10). The average absolute value of the error for tasks 2 and 3 was 20%. Additionally, the average calculated result is very close to the actual weight for all three tasks. The full results of the test are expressed below in Figure 1, which expresses the distribution of value of the errors by percentage for all three tasks against the correct outcome of 1, 5, and 10, respectively:

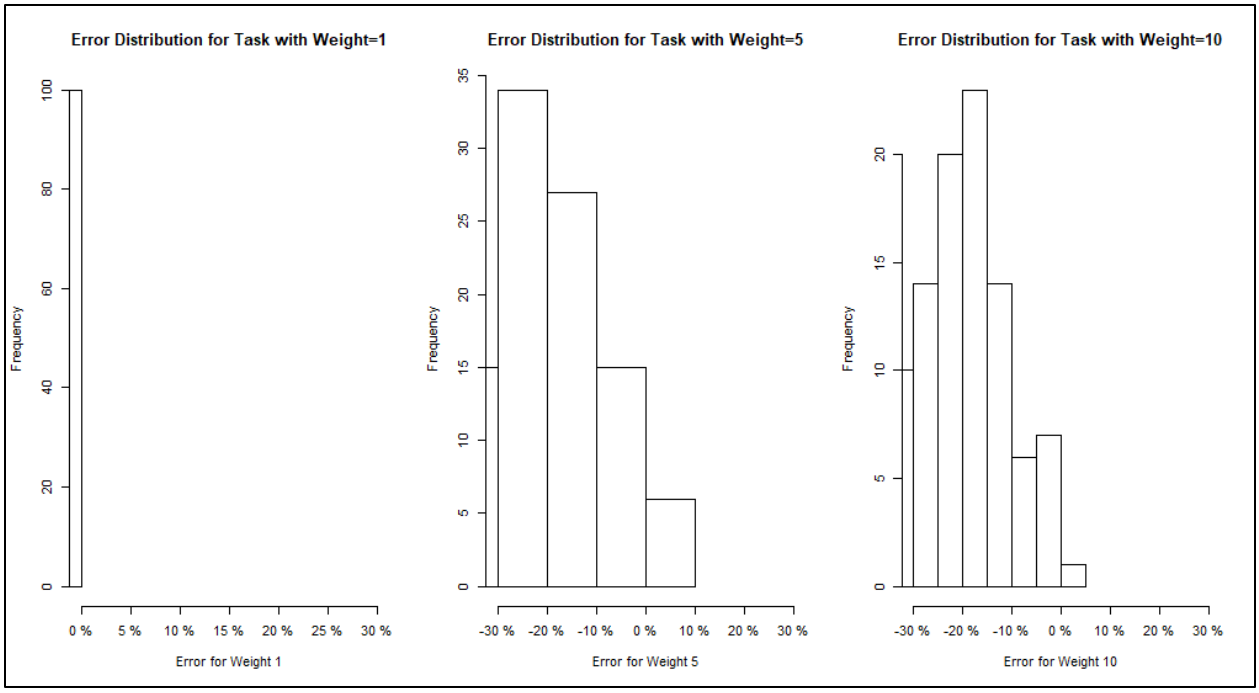


Figure 1

Figure 1 shows that the task with the lowest weighting of 1 was always accurate, suggesting that the analysis always correctly identified the lowest weighted task. Figure 1 also illustrates that the error for tasks 2 and 3 is almost always negative. Since the test was gradually increasing the weightings for tasks 2 and 3, these results suggest that the analysis generally terminates before the accurate weightings are found. A likely explanation is that adjustments are only made if the p-value for the *r* coefficient is less than 0.05. This standard is typically used in hypothesis testing to minimize the probability of producing false positives while still preserving power for the hypothesis test. However, in this analysis the objective is to obtain the most accurate results possible, not to avoid a false positive result. Therefore, it is possible that relaxing the p-value requirement will result in more accurate results. To test this theory, we will relax the p-value requirement from 0.05 to 0.5 and will rerun the test. Using a p-value of 0.5 will have the effect of always maximizing the likelihood of improving the weighting while minimizing the likelihood of overcorrecting the weighting since the weighting will always be

adjusted if there is more than a 50% chance that adjusting the assigned weight will improve its accuracy. The code for this analysis and test can be found in Section 2 of the Appendix. The results of are expressed below in Table 5:

Task Names	Actual Weighting	Average calculated weighting	5th Percentile of calculated weighting	95th Percentile of calculated weighting	Average Absolute Value of Error
Task 1	1	1	1	1	0
Task 2	5	4.9	3.9	6.3	13%
Task 3	10	9.7	7.7	12.4	13%

Table 5

The full results of the test are expressed below in Figure 2:

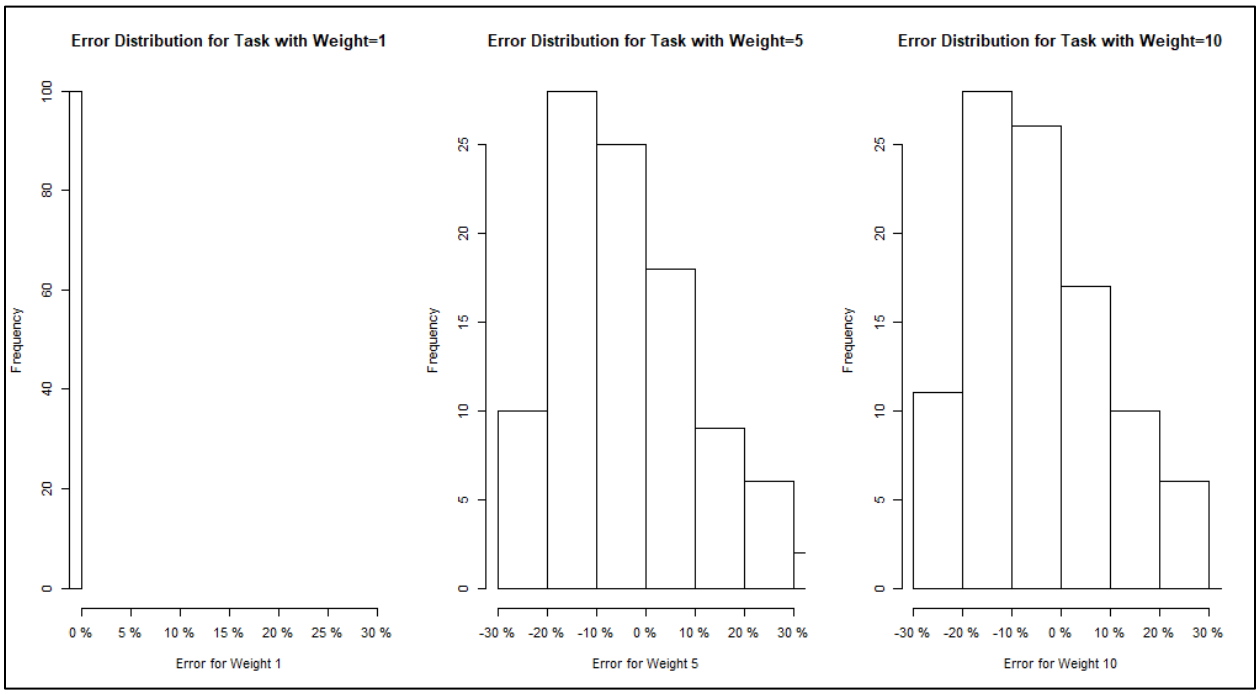


Figure 2

As Table 5 and Figure 2 illustrate, this updated approach with a less restrictive p-value threshold yields more accurate results. The low-end outliers are closer to the actual task weightings and the average absolute value of the error was reduced from 20% to 13%. To

determine if the analysis can detect smaller differences in weightings, we will run the analysis again using data generated with actual task weights (1,2,3). Since using the p-value of 0.5 instead of 0.05 yielded more accurate results, on average, we will use the same approach for this test. The code for this analysis and test can be found in Section 2 of the Appendix. The results are expressed below in Table 6:

Task Names	**Actual Weighting**	**Average calculated weighting**	**5th Percentile of calculated weighting**	**95th Percentile of calculated weighting**	**Average Absolute Value of Error**
Task 1	1	1	1	1	0
Task 2	2	2	1.9	2.1	2.5%
Task 3	3	3	2.8	3.1	2.2%

Table 6

The full results of the test are expressed below in Figure 3:

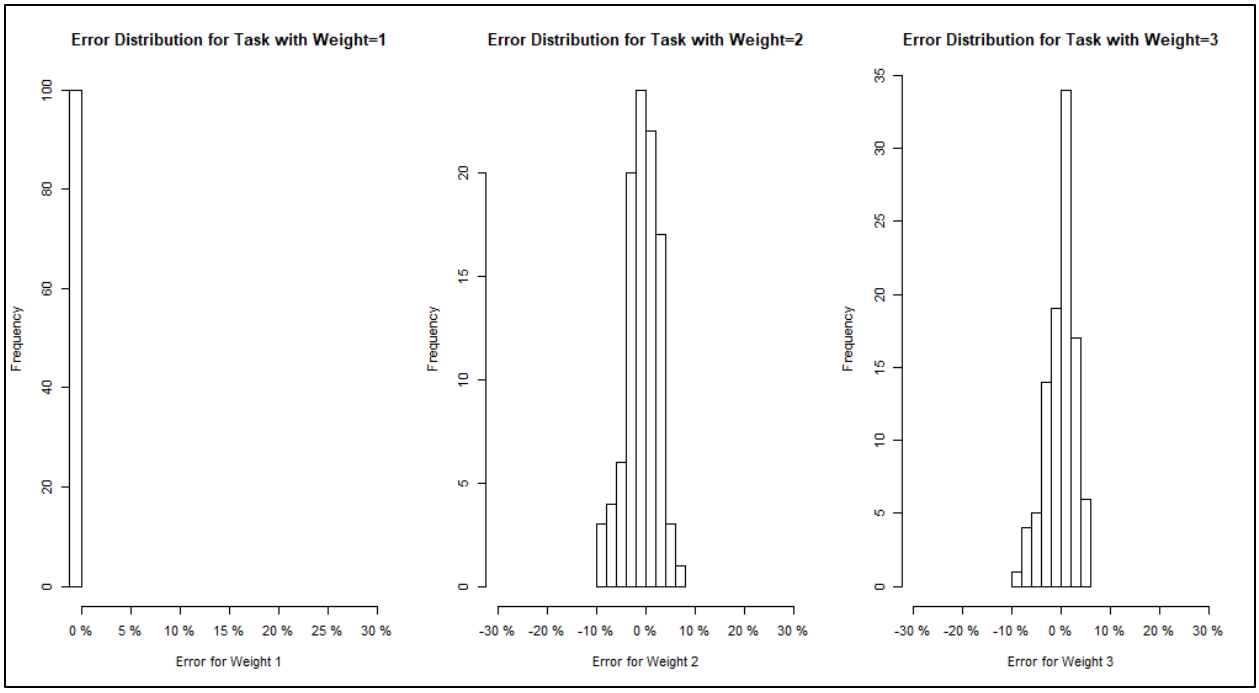


Figure 3

As Table 6 and Figure 3 illustrate, the method works even better when the differences between the weights are smaller. One limitation of this analysis is that it assumed that each performer is equally productive. While the method works well under these conditions, these conditions are unfortunately rarely found in real-world scenarios.

Capacity Weighting Correlation Analysis – Adjusted for Productivity

To simulate a more realistic scenario, the next test will be run with data that simulates varying productivity. To simulate varying productivity, each performer will be randomly assigned a productivity coefficient. To simulate commonly observed productivity distributions, these coefficients will be assigned using a normally distributed random variable with $\mu = 100$ and $\sigma = 15$. This productivity coefficient will be used as each performer's starting capacity instead of using 100 for all performers. We will use actual task weights (1,2,3) and will use 50

performers. The code for this analysis and test can be found in Section 2 of the Appendix. The results are expressed below in Table 7 and Figure 4:

Task Names	Actual Weighting	Average calculated weighting	5th Percentile of calculated weighting	95th Percentile of calculated weighting	Average Absolute Value of Error
Task 1	1	1.0	1	1.2	3.7%
Task 2	2	1.7	1	3.0	31.4%
Task 3	3	2.2	1.25	4.0	33.9%

Table 7

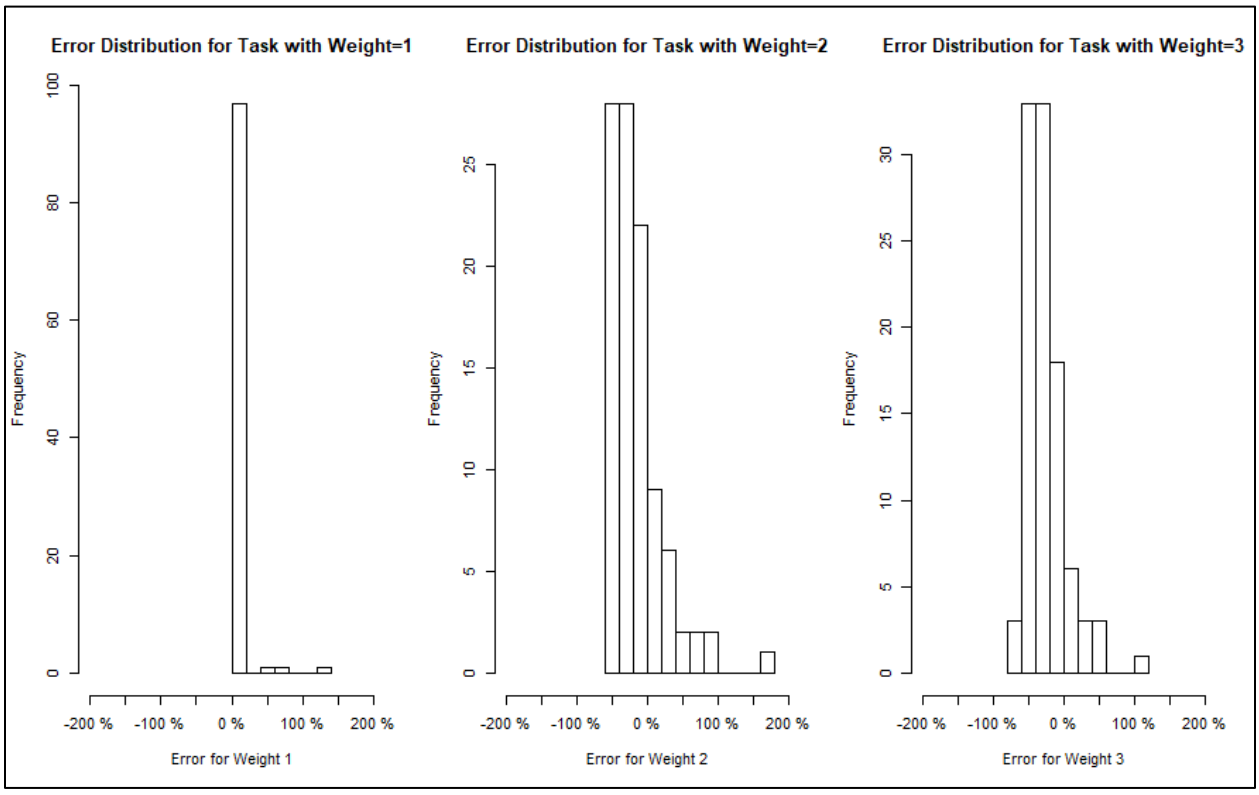


Figure 4

Table 7 and Figure 4 show that the Capacity Weighting Correlation Analysis does not perform as well when the productivity varies by performer. To improve the accuracy of the method when productivity varies, we will need to determine a way to estimate the productivity of each performer. If the productivity of each performer can be reliably estimated, the varying productivity can be accounted for by dividing the number of each items that each performer completed by the productivity coefficient that corresponds to that performer. This step would have the effect of compensating for the different productivities which should allow the test to perform similarly as to when the productivities do not vary.

One simple method for estimating productivity would be to calculate a count of unweighted tasks completed. Since we know each task has a true weight that differs from one another, this method would likely be unreliable. It is worth noting in Table 7 above that while the estimated weights are not perfectly accurate, they are directionally correct. That is, the weights for the tasks that take longer to complete tend to be estimated as heavier and the weights for the tasks that take less time to complete tend to be estimated as lighter. These weights could therefore be useful in estimating productivity differences. We will estimate productivity differences by calculating total weighted scores using the weights we originally calculated by running the Capacity Weighting Correlation Analysis once. The number of items completed by each performer could then be divided by the productivity coefficients to account for the varying productivity.

This method presents one problem: the total weighted scores will all be reduced to the same number meaning there will be no variation among total weighted scores. If there is no variation among total weighted scores, it will be impossible to calculate a correlation coefficient as zero variation will lead to a zero in the denominator of the equation for the Pearson

Correlation Coefficient. To account for this problem, we simply need to use two months of performance data. The Capacity Weighting Correlation Analysis will be applied to the first month's data to determine starting weights to estimate the productivity of each performer. In the second month, the number of each item produced by each performer will be divided by the estimated productivity coefficients calculated from the first month's performance. This adjusted data from the second month will then be used in the final Capacity Weighting Correlation Analysis.

To test this two-month method to account for differing productivities, we will produce two months of randomly generated data and apply the method. We will then run the method 100 times with 50 performers in each trial and actual task weights (1,2,3). The code for this analysis and test can be found in Section 2 of the Appendix. The results are expressed below in Table 8 and in Figure 5:

Task Names	**Actual Weighting**	**Average calculated weighting**	**5th Percentile of calculated weighting**	**95th Percentile of calculated weighting**	**Average Absolute Value of Error**
Task 1	1	1.	1	1	0%
Task 2	2	1.9	1.5	2.3	11%
Task 3	3	2.7	2.2	3.3	12%

Table 8

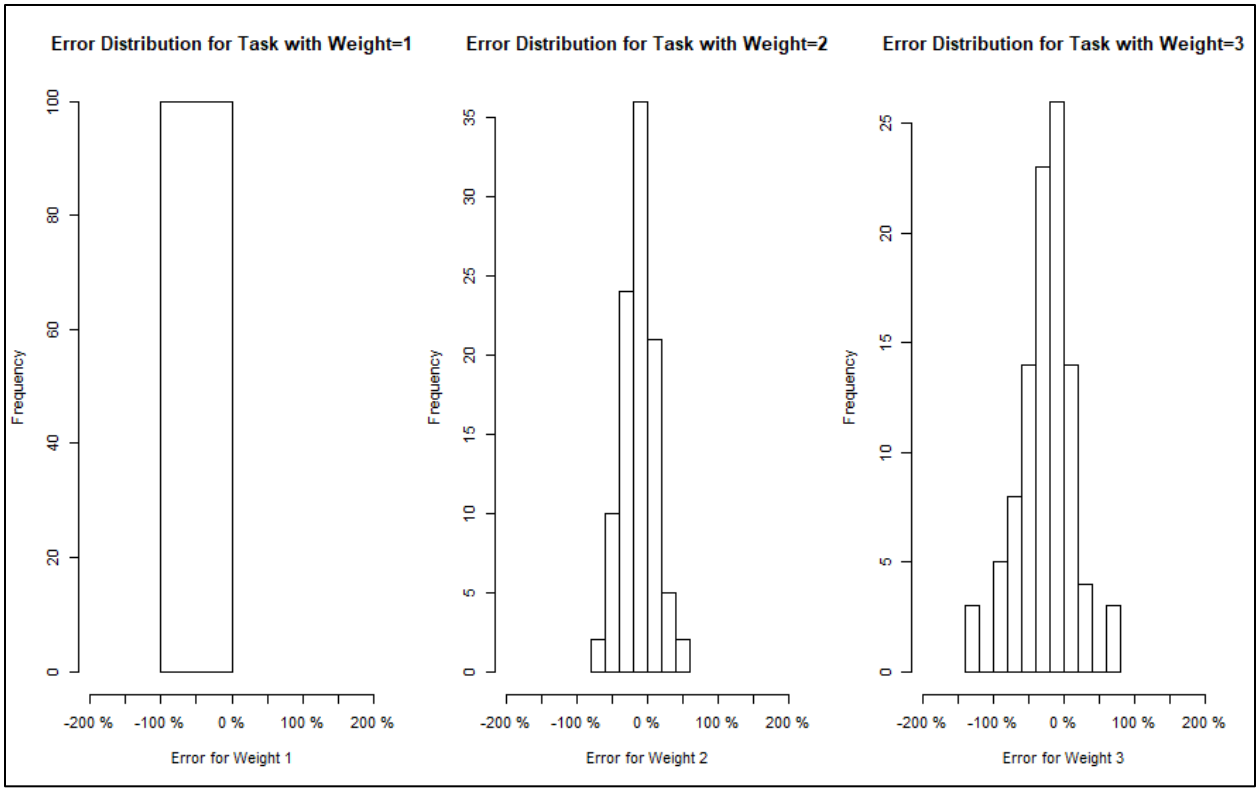


Figure 5

Table 8 and Figure 5 demonstrate that this enhanced analysis offers an improvement over the one-month analysis because it accounts for the differing productivities and yields an acceptably low error rate.

II. Performance Consistency Analysis

Analysis Background

The Capacity Weighting Correlation Analysis offers a valuable tool for accurately measuring performance but even if performance can be reliably and accurately measured, a potential performance metric may not meet the third criterion discussed in the introduction: the metric is directly impacted by the performance of the individuals. Consider, for example,

turnaround time metrics. Many businesses are interested in how long a task takes to perform. Shorter tasks often contribute to lower expenses, which benefit shareholders and drive greater customer satisfaction. Shorter turnaround times can also be a comparative advantage that allows businesses to differentiate themselves in a competitive environment. Turnaround times, however, are often impacted by many factors. If an individual is responsible for processing a specific request, the time it takes to complete the request may be impacted by the employee but may also be impacted by the complexity of the request, dependencies on external parties or information, responsiveness of the customer or vendor, and many other factors that the employee cannot control. Assigning full accountability for the entire process to the employee could lead to reduced morale and, in some cases, could create an adverse incentive. For example, an employee may rush a customer or be tempted to take shortcuts that could compromise quality. Ensuring an employee has sufficient control over the outcome of a metric is therefore critical to effective performance measurement.

Ideally, external factors would be controlled and accounted for in a performance metric. Unfortunately, this is often not possible. External factors, such as complexity, might be difficult to quantify. Furthermore, the number of tasks an employee completes in a measurement period might be insufficient to allow for statistical control of external factors. If external factors cannot be controlled, it is important that a business confirm that the employee's influence over the outcome is sufficient to merit using the metric to evaluate each employee's performance. This raises several questions, including "How much influence is sufficient?" and "How can the employee's influence on the metric be measured?".

Common Approaches to Measuring Performance Consistency

A common approach to determining an employee's influence on a metric is to build a statistical model that includes the outcome of the metric as the dependent variable. Each employee can be coded as a discrete dummy variable in the model and other factors that may contribute to the outcome can be included as either continuous or discrete independent variables. This common approach presents several challenges. First, in a large team coding each employee as a discrete independent variable can make a model excessively large and complex. Many common statistical software packages limit the number of discrete groups that can be analyzed, making these methods obsolete for large teams. Additionally, many of the external factors, such as complexity of the task being performed, are difficult to quantify and therefore may be difficult to include in the model.

A simple and common version of this modeling approach is an Analysis of Variance (ANOVA) model where the output of the metric, such as cycle time, is grouped based by employee. The ANOVA will allow an analyst to determine, at some pre-determined level of significance, whether the average outcome truly differs by employee. The challenges with this approach are numerous. First, the test often yields the unsatisfying result that some performers have the same average while others differ from one another. It is difficult to interpret such results as they do not indicate whether some performers are consistently better or worse than others. Additionally, this test's statistical power to detect a difference between performers is limited unless a large sample size of performance data can be gathered over time, which is often infeasible as attrition, special projects, and other factors limit the ability to gather consistent data. Even if a large enough sample size can be gathered and the ANOVA reveals that many of the average performances are truly different, these differences might only be discernible with many

months of data. The differences may also be so small that they are not practically significant for performance measurement purposes. All of these challenges limit the usefulness of the ANOVA when evaluating a potential performance metric.

Performance Consistency Analysis – Quartile Based Method

An analytical method designed to solve this problem would be more useful if it required less data and provided more confidence that the employee truly impacts the outcome, not just that the results differ by employee in the long run. We will develop such a method, which we will call the "Performance Consistency Analysis". We will begin by using mathematical and statistical principles to formalize logical hypotheses. A simple hypothesis regarding performance consistency is that if employees are able to impact the outcome of a metric, one would expect consistency in the employees' performance relative to one another. While external factors might affect each individual's performance each month, the high performers would tend to perform well consistently and the low performers would tend to perform poorly consistently.

This trend of consistency is important for another reason beyond identifying each employees' ability to impact the outcome of a metric. Even if an employee can impact the outcome of a metric, the metric will only be useful as a performance metric if the employee can *consistently* impact the metric's outcome. Otherwise, coaching and managing to the metric on a regular basis will be very challenging considering that top and bottom performers will change each month. Being able to produce and measure consistent performance results is fundamental to any performance measurement system.

To formalize the concept of consistent performance across performers, we first need to devise a system to numerically represent the performance in each month. A simple version of such a system divides all performances in any given performance period into quartiles. This

approach could be improved upon as it sacrifices information by making no distinction between the performances within each quartile or taking advantage of measuring the variability between each performer. Notwithstanding, this simple approach offers a starting point for formalizing the rankings of the performers for formal evaluation and comparison.

Once the performances for two months are divided into quartiles, the results must be compared. A simple form of comparison is to determine what percent of the performers fell into the same quartile in both performance periods. If the performers have no impact on the outcome of the metric, then what quartile each performer is assigned to each month will be determined entirely by external factors. The distribution across the quartiles will therefore appear to be random. We could expect that, on average, 25% of the performers would end up in the same quartile in both performance periods. If the performers have a strong influence on the outcome of the metric, we would expect greater than 25% of the performers to fall in the same quartile in both performance periods, indicating that the performers are more consistent. To make a statistically robust comparison, however, we must understand not only the expected value but the probability distribution of the proportion of performers falling in the same quartile in both periods.

Random events with two outcomes can be modeled using a binomial distribution. The binomial distribution models the expected number of successes in n trials. If the outcomes are divided by n, the distribution can be used to model the expected probability of a successful outcome. In the Performance Consistency Analysis, a successful outcome is defined as a performer falling in the same quartile in two consecutive months. As previously mentioned, the expected value of the probability of such an event occurring is 25% since there are four quartiles.

When the number of trials is sufficiently large, a binomial random variable can be approximated using a normal distribution as illustrated in (2)

$$P \cong N(\mu = p, \sigma = \sqrt{\frac{p(1-p)}{n}}) \quad (2)$$

where *P* is the probability of an event occurring, *p* is the expected value of the probability of the event occurring, and *n* is the number of trials. The probability distribution of the number of the *n* performers falling into the same category can therefore be approximated with the distribution illustrated in (3):

$$P \cong N(\mu = 0.25, \sigma = \sqrt{\frac{0.25(1-0.25)}{n}}) \quad (3)$$

While the normal approximation of the binomial distribution is a well established method, it will be helpful to test this method before using it to ensure that it is sufficient for use in this model. To do so, we must first generate data for testing. We will create a simulation that will randomly assign performance scores to a group of *n* performers for two periods. The performers will then be ranked according to those scores in each period and will be grouped into quartiles based on their performance ranking. The quartiles will then be compared and the proportion of performers falling in the same quartile in both periods will be recorded. This simulation will be run 1,000 times for *n* performers where $n = (1,2,\ldots 100)$. The 95^{th} percentile of the returned proportions will be calculated for each *n* and compared to the expected 95^{th} percentile calculated using the normal approximation. The code for this simulation can be found in Section 1 of the Appendix. The differences between the simulated and theoretical results are illustrated in Figure 6:

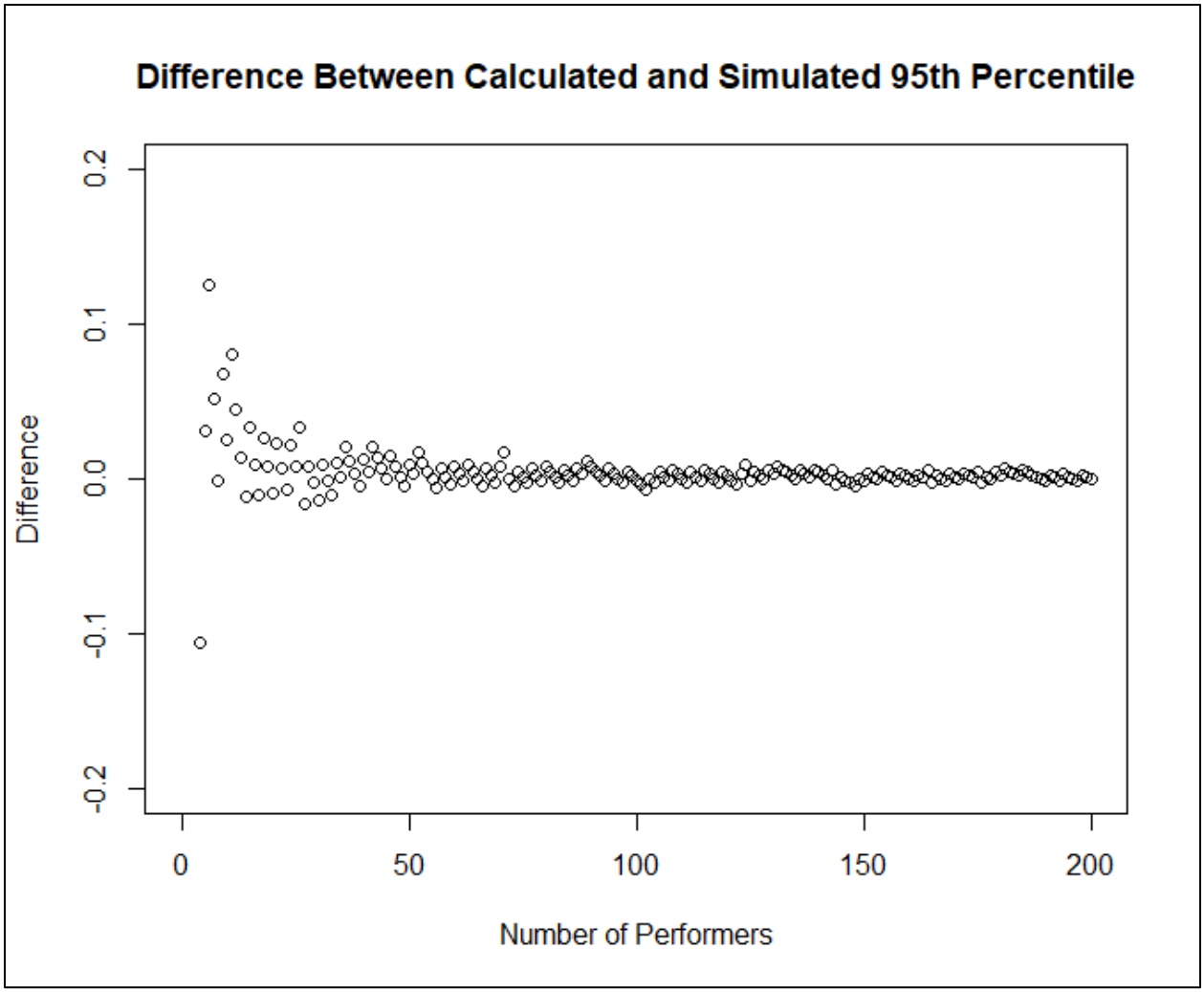


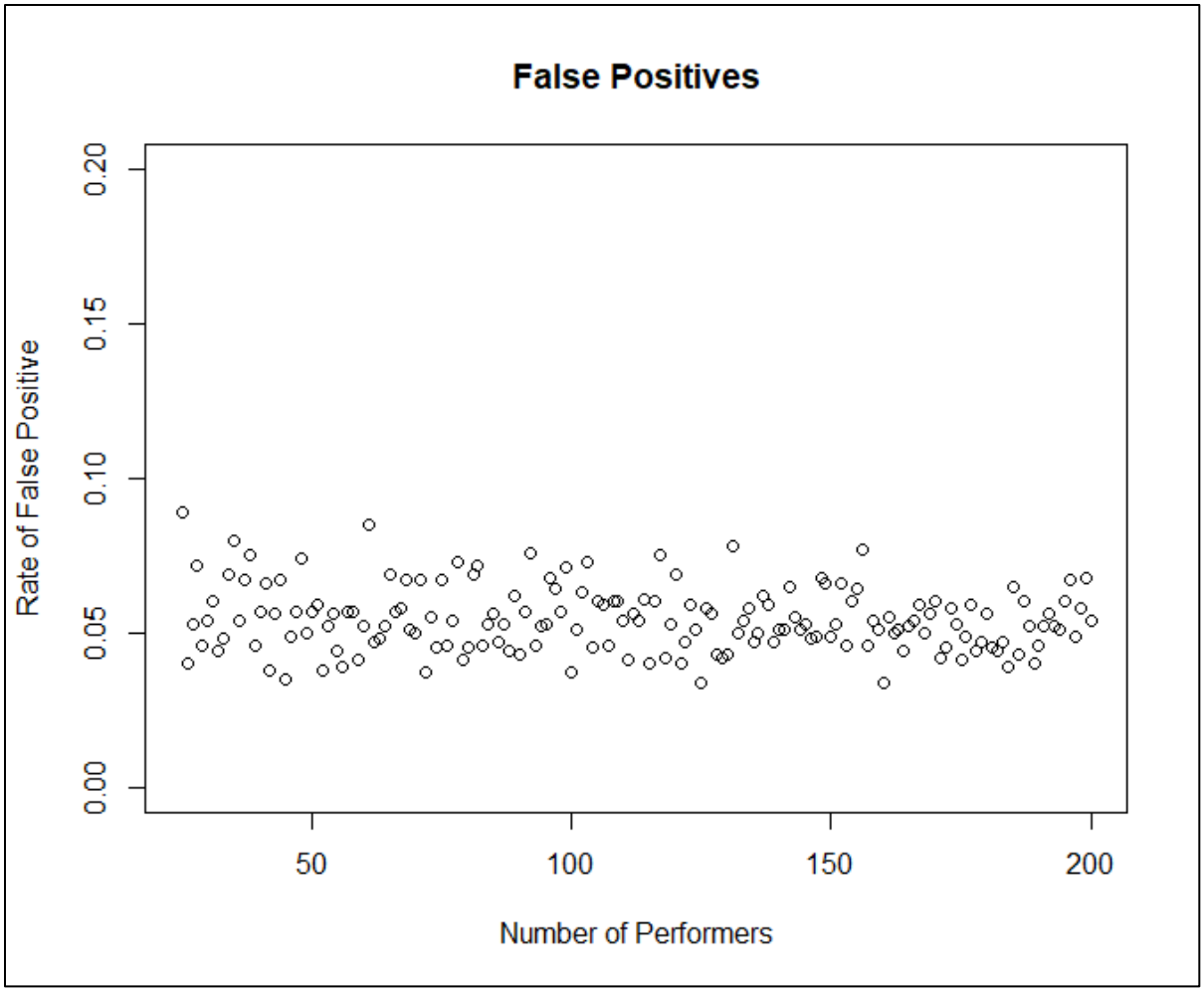
Figure 6

As figure 6 illustrates, the difference between the simulated and theoretical 95th percentiles are sufficiently small when the number of performers is 25 or more. We can therefore use the normal approximation of the binomial variable to calculate a 95% one-sided confidence interval for the value of *p*. Using this idea, (4) calculates a threshold for evaluating whether the observed proportion of performers in the same quartile in two months could result from a random assignment of quartiles in both months. This formula uses a Z critical value of 1.645 which yields a 95% confidence level for a one-sided test: (4)

$$.25 + (1.645 \times \sqrt{\frac{0.25(1-0.25)}{n}})$$

where *n* is the number of performers and 1.645 is the z-score for the 95th percentile of a one-sided normal distribution. Since only 5% of the results derived from a random reordering of quartiles will exceed this threshold, if a proportion larger than this threshold is observed, we can conclude at the 5% level of significance that the observed results were not derived from random chance.

With the threshold established, the test can be performed by calculating the proportion of performers who fall into the same quartile in two periods and comparing this proportion to the established threshold. This test will serve as the first version of the Performance Consistency Analysis. First, to confirm that this test delivers a true alpha of 0.05, that is that the test only carries a 5% probability of committing a type I error, we will run the simulation 1,000 times for each *n* where *n* = (25, 26,…200) and all performances are randomly assigned. Since all performances are randomly assigned, any successful result will be a false positive. We will then plot the rate of false positive as a function of the number of performers. The rate of false positives should be approximately 5% for any number of performers. The code for this simulation can be found in Section 1 of the Appendix. The results are illustrated below in Figure 7:



Mean = 5.4%

Figure 7

As figure 7 illustrates, the test generally yields a true alpha of 0.05. With the probability of committing a type I error established, the power of the test can be analyzed. Determining the true power of the test is challenging since the results will be influenced not only by whether the performers influence the outcome of the metric, but also by how much the performers influence the outcome of the metric. For example, performers who influence the outcome of the metric but yield tightly distributed performances will be much harder to detect than performers who influence the outcome of the metric and have widely distributed performances. To address this problem, we will establish a standard for evaluating the power of the test.

Simulating a completely random distribution of performances can be accomplished by assigning a random score between 1 and 100 to each performer then sorting the performers according to the score and assigning the appropriate quartile. This method is equivalent to the method used above to verify the p-value of the test. Establishing a non-random performance distribution is much more challenging. Once the random element is created, the mix of random and non-random elements can be incorporated into the overall performance outcome by assigning coefficients to the random and non-random elements. These coefficients will sum to 100%. This approach will result in each performance being a weighted average of the random and non-random effects. This approach, however, still presents the challenge of consistently simulating the non-random element of performance.

A simple approach would be to create an evenly spaced distribution of performances across all performers. This approach is, unfortunately, unrealistic and presents multiple problems. Performances typically cluster around some point and spread out to one or both extremes. The evenly spaced approach would not account for this commonly observed pattern. Another question this approach presents is what range should be applied to the distribution of non-random performances. The distribution could be narrower, as wide, or wider than the distribution of random performances. These questions bring to light that it would be a mischaracterization to claim that performance outcomes are a quantifiable blend of random and non-random elements. Claiming that a performance is made up of y% non-random factors and (100-y)% random factors undermines the fact that the range and distribution of each type of factor have a strong effect on the results. Notwithstanding, creating a standard for the mix of effects is necessary to test different approaches for detecting and measuring the presence of non-random factors.

While the distribution of non-random factors will follow a variety of distributions depending on the task being performed as well as the performers completing the task, a normal distribution represents a typical performance distribution reasonably well and will therefore be used for the standardized non-random portion of the test. Since over 99.7% of the data in a normal distribution falls within 3 standard deviations of the mean, the non-random element of performance can be calculated by applying the inverse cumulative distribution function to a normal curve that is centered at 50 with a standard deviation of $^{100}/_{6}$. That is, the non-random performances of *n* performers can be calculated using the approach illustrated in Table 9:

Performer 1	InverseCDF($100 \times {}^{1}/_{n}$, N(μ=50, σ=17.5))
Performer 2	InverseCDF($100 \times {}^{2}/_{n}$, N(μ=50, σ=17.5))
Performer 3	InverseCDF($100 \times {}^{3}/_{n}$, N(μ=50, σ=17.5))
…	…
Performer *n*	InverseCDF(100×1, N(μ=50, σ=17.5))

Table 9

The random element of the performance can be calculated by generating a random number between 1 and 100. The two elements will then be combined using the weighted average method described above and expressed below in (4):

$$NonRandomElem(y\%) + RandomElem(1 - y)\% \quad (5)$$

where *y* represents the percentage of the performance that is determined by non-random factors, $NonRandomElem$ represents a non-random factor and $RandomElem$ represents a random factor. While the testing data could be structured other ways, using this standard approach will enable an evaluation of different methods to determine which method is most powerful. Unfortunately, formally measuring the power of any method can only be accomplished for any

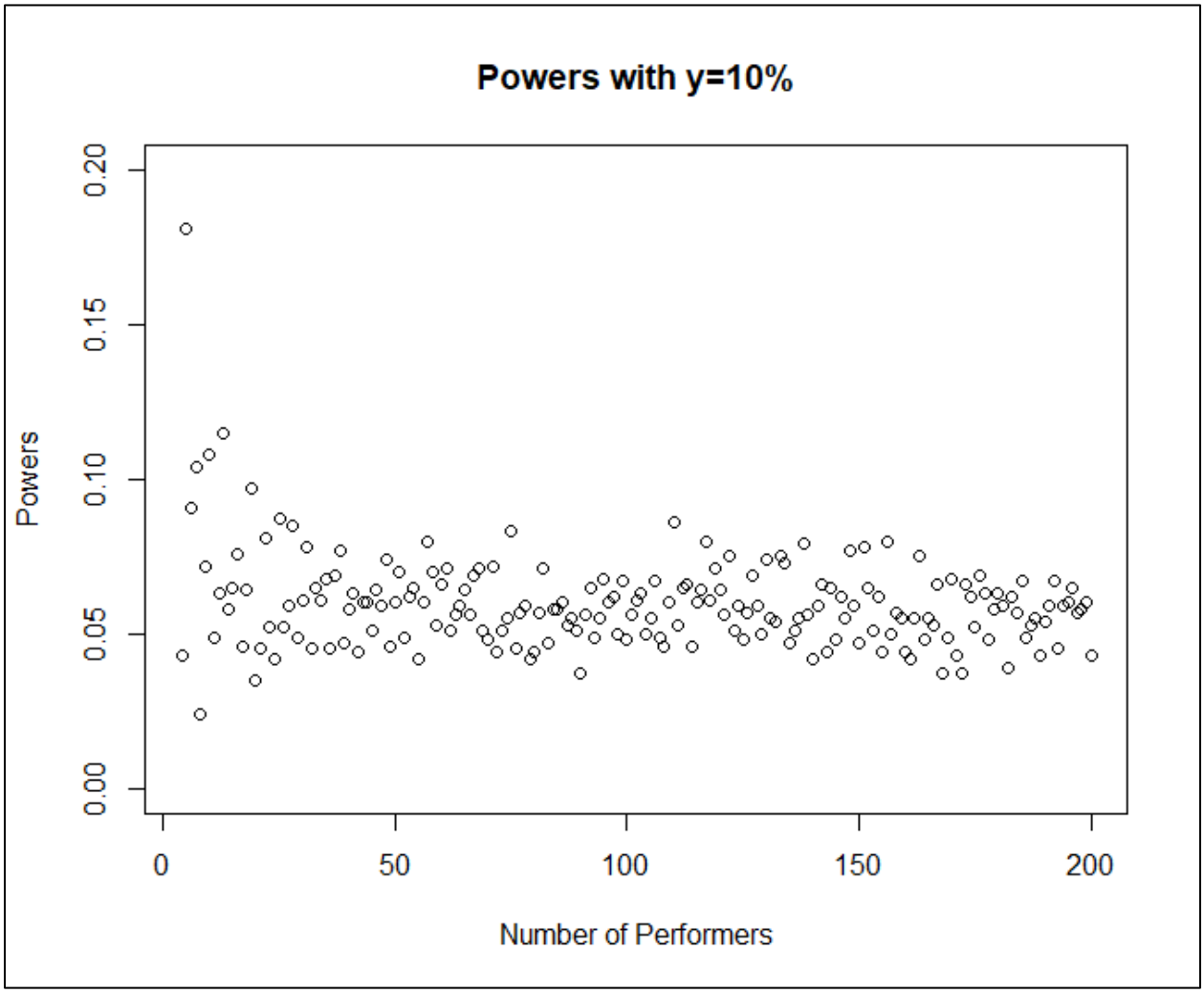
given standardized approach, such as the one outlined above. For example, a test can be said to be able to correctly reject a null hypothesis x% of the time when *y*% of the performance is non-random and (100-*y*)% of the performance is random when the non-random elements of the performance follows a strictly prescribed pattern. The "when the non-random elements of the performance follows a strictly prescribed pattern" disqualifier unfortunately cannot be removed because it would introduce the problem of there being an infinite number of ways for the non-random element of the performance to be defined. Notwithstanding, this formal method will provide a suitable platform for development and testing of a method to detect and measure the non-random performance elements.

Using this formal method, the previously described Performance Consistency Analysis using the quartile comparison method can be evaluated. To begin the analysis, performance data will be generated for a group of hypothetical performers. For each performer a random score will be determined by randomly assigning a number between 1 and 100. A non-random score will be assigned using the Normal CDF method described above. The two elements will then be combined according to (5).

The results can then be evaluated using the Performance Consistency Analysis that will calculate the proportion of performances that fell in the same quartile in two different performance periods and compare the results to the comparison threshold calculated in (4) above. This method introduces two variables: the proportion of the performance determined by non-random elements (expressed by *y*) and the number of performers (expressed by *n*). For the initial evaluation, we will hold *y* constant at 10% and will evaluate the power of the test for *n* = (4, 5,…200). The test will be run 1,000 times for each *n.* The code for this test can be found in section 1 of the Appendix. The details of the test are summarized below in Table 10 and Figure 8:

Test objective:	Correctly reject the null hypothesis
Null Hypothesis:	The performance results are completely randomized: $y = 0$
Performance results:	$NonRandomElem(y\%) + RandomElem(1 - y)\%$
Number of performers (n):	Variable
Influence of random elements (y):	10%

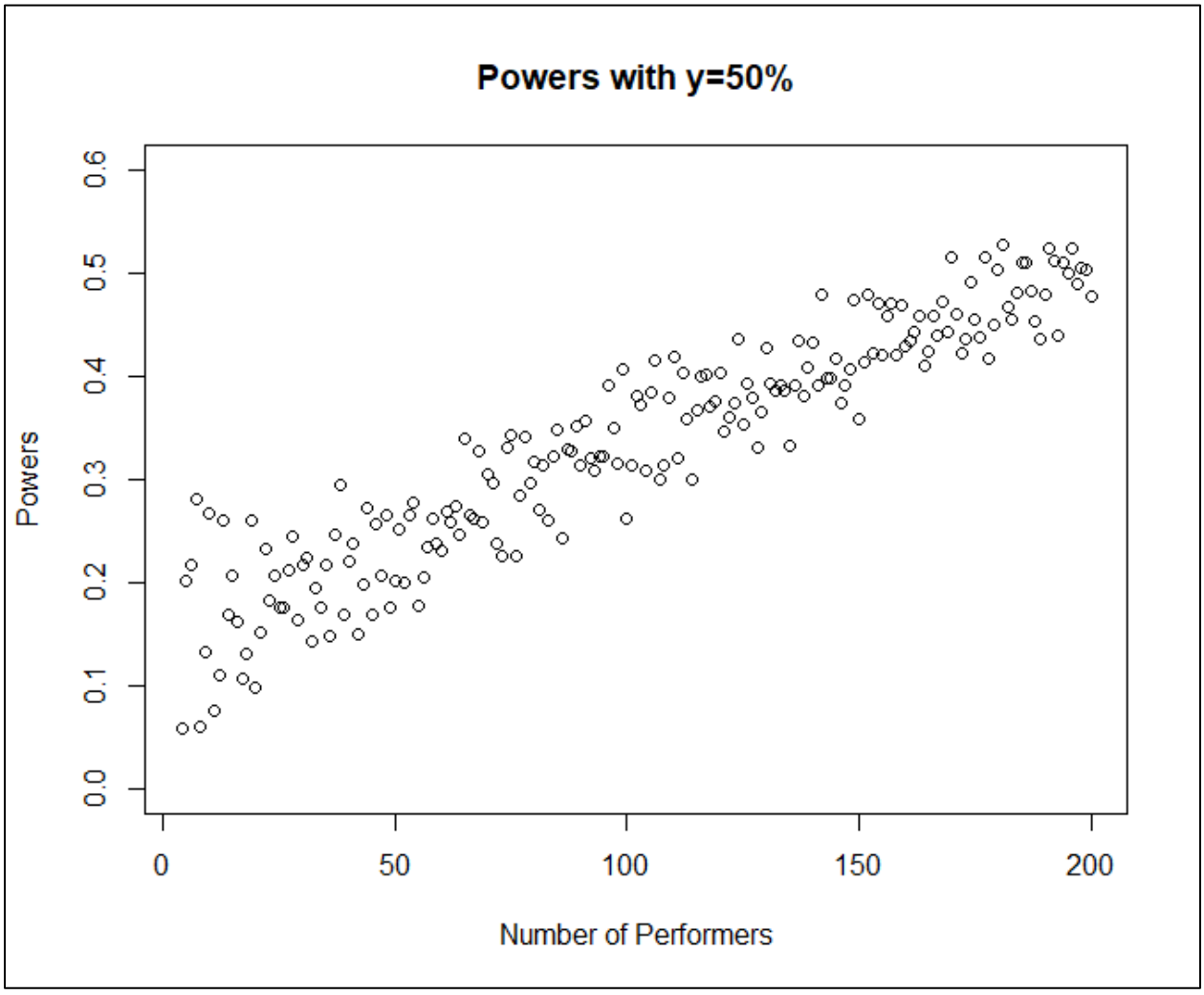
Table 10



Mean = 0.06

Figure 8

This analysis is too weak to detect the influence of the non-random factors at y=10% because it only correctly identifies a difference about 6% of the time. Too few performers fell in the same quartile in 95% of the trials and the proportion of total performers who met this criterion were therefore below the comparison threshold. The analysis will be tested again to determine if it can detect a much larger difference of y=50%. The results are illustrated below in Figure 9:



Mean = 0.34

Figure 9

The analysis is much more powerful when half of the performance is determined by random factors. The analysis correctly rejected the null hypothesis $^1/_3$ of the time, on average, and $^1/_2$ of the time when 200 performers were evaluated. Notwithstanding, an analysis with a power of $^1/_3$ only offers limited usefulness as it would fail to detect a difference more than half of the time. A more viable solution would correctly reject the null hypothesis at least 50% of the time when 50 or more performers are used and the non-random effects are truly greater than 0. The ability for the quartile version of the analysis to reliably detect a difference is hindered by several factors.

Performance Consistency Analysis – Vector Based Method

The most significant limiting factor of the quartile version of the analysis is that organizing the data into quartiles causes a loss of information. Ranking and distance within each quartile are completely ignored by this analysis. To avoid losing information by reducing the results to quartiles, the performers could simply be ranked. That is, the performers could be listed in order of their performance and then assigned numbers according to their ranking from 1 to *n* where *n* is the number of performers. To evaluate how much each performer moved from one month to the next, the difference of the two rankings could be calculated and squared for each performer. The squared differences could then be averaged to represent the total amount of movement from one month to the next. More movement would lead to a larger mean of squared differences. If the performers have a strong influence on the outcome of the metric, we would expect the rankings to be more similar from one month to the next. In this case, the mean of squared differences would be smaller than if the performance were completely random.

Therefore, we can use a small mean squared difference to identify when the performance is relatively consistent and the performers therefore likely impact the outcome of the metric.

To make this enhanced version of the Performance Consistency Analysis viable, we must develop a method to calculate a comparison threshold for the mean squared differences for any number of performers *n*. To do so, the expected distribution of the mean squared differences for the completely random case must first be found. This problem can be approached by considering a vector of size *n* that contains ordered integers from 1 to *n*. A second vector is then generated by randomly reordering the entries of the original vector. The mean squared differences between the two vectors is then calculated.

We will begin by finding the mean of this distribution. First, we will consider every possible outcome of the squared differences, beginning with the largest. The largest possible squared difference is $(n-1)^2$. This could occur two ways: a performer moving from the first to the *n*th position and a performer moving from the *n*th to the first position. The next largest possible squared difference is $(n-2)^2$. This could occur four ways: a performer moving from the first to the (*n*-1)st position, a performer moving from the second to the *n*th position, a performer moving from the (*n*-1)st to the first position, and a performer moving from the *n*th to the second position. This pattern is developed further in Table 11:

Squared Difference Value	**Number of Possible Outcomes**	**Value × Number of Outcomes**
$(n-1)^2$	2	$2(n-1)^2$
$(n-2)^2$	4	$4(n-2)^2$
$(n-3)^2$	6	$6(n-3)^2$
…	…	…
$(1)^2$	2(*n*-1)	$2(n-1)(1)^2$

Table 11

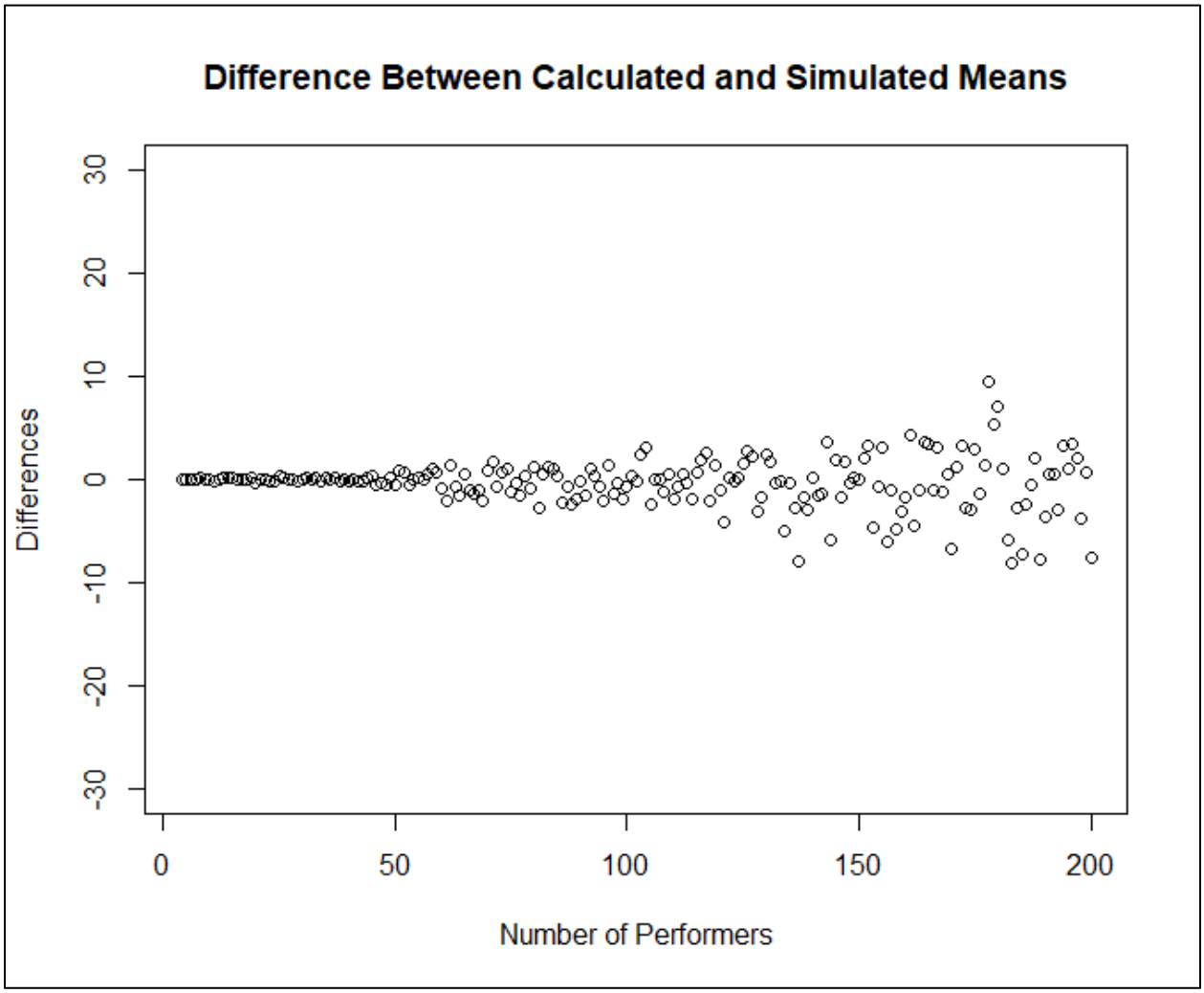
The final column of the table outlines every possible value multiplied by the number of ways it could occur. The sum of this final column represents the sum of all possible outcomes. This result is succinctly expressed in the general form in (6):

$$\sum_{i=1}^{n-1} 2i(n-i)^2 \tag{6}$$

This formula calculates the sum of each possible squared difference for each item in the randomly reordered vector. Calculating this sum is the first step to obtaining the expected value of the distribution. For each element in the vector, there are *n* possible outcomes and each of these outcomes is equally likely since the vector is randomly reordered. Therefore, to calculate the expected sum of the squared differences, we must divide the result in (6) by *n*. Since there are *n* elements in the vector, to calculate the mean of the squared differences we must divide by *n* again. The mean of the sum of squared differences for a randomly reordered vector is therefore expressed in (7):

$$\frac{\sum_{i=1}^{n-1} 2i\,(n-i)^2}{n^2} \tag{7}$$

To test this formula, we will create a simulation by randomly reordering vectors of length 4 to 200. The vectors will be made up of integers that correspond to their position in the vector. For example, the vector of length 4 will be (1, 2, 3, 4). The vector of length 200 will be (1, 2, 3, 4, …, 200). Each vector will be randomly reordered 10,000 times and the mean of the squared differences between the cardinally ordered and randomly ordered vectors will be calculated. The mean of these means of squared differences will then be compared to the expected mean calculated by (7). The difference between the expected and observed result will then be calculated and expressed in a histogram. The code for this simulation can be found in Section 1 of the Appendix. The results of this simulation are expressed below in Figure 10:



Mean of Absolute Value of Errors = 1.57

Figure 10

As figure 10 illustrates, the formula to estimate the mean of the squared differences is accurate. Figure 10 also suggests that the variation of the mean of squared differences increases as the vector length increases but the difference is still so small that it can be ignored because it will not materially impact the results of the analysis. Next the variation around this mean must be estimated. This process will begin by observing a distribution of 1,000 means of squared

differences from 1,000 simulations of randomly reordered vectors of length 100 as shown below in Figure 11:

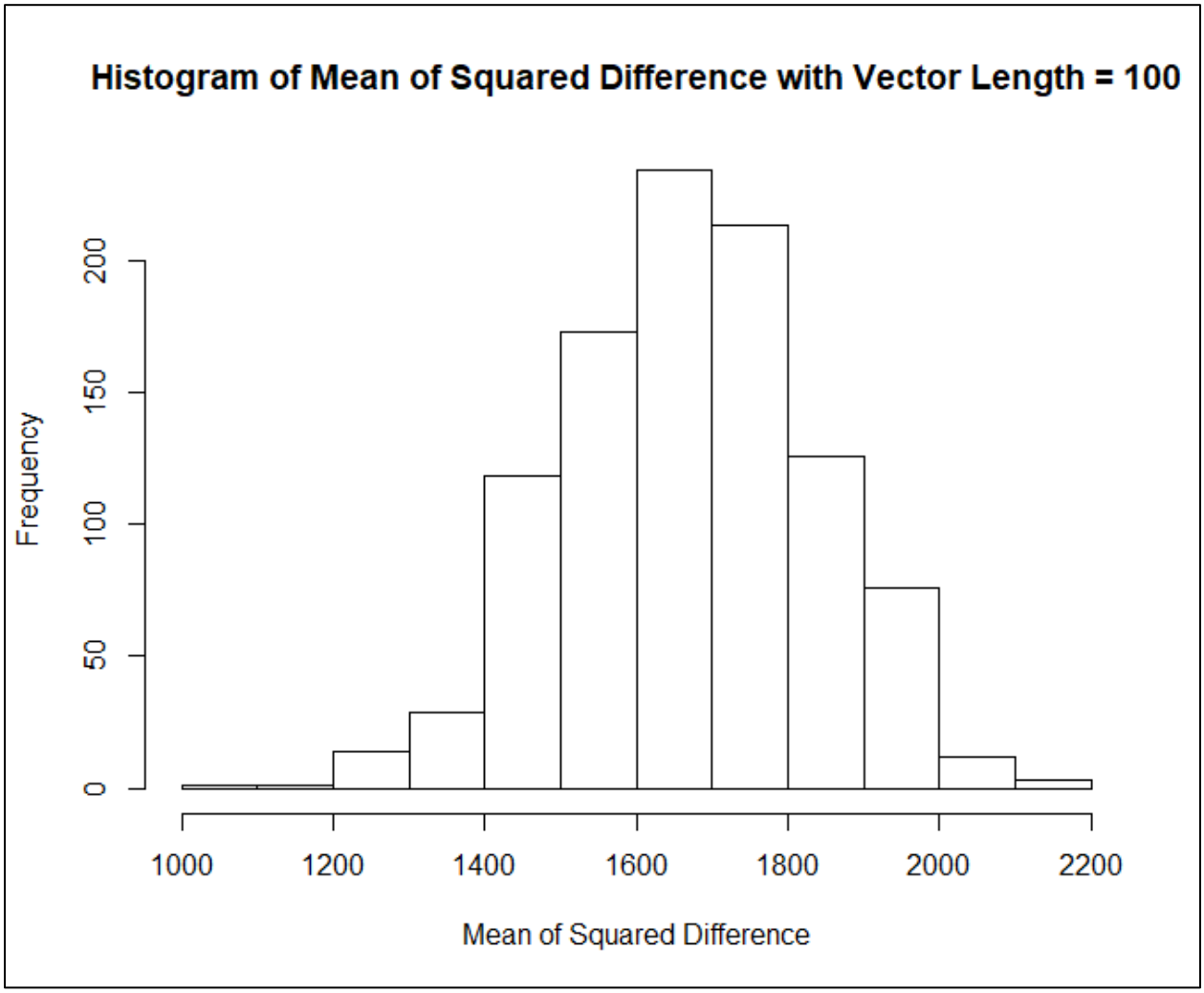


Figure 11

As Figure 11 illustrates, the distribution appears to be normal. If this is the case, we only need to determine how to calculate the standard deviation in order to determine the distribution. The mean can be calculated using (7), which sums every possible outcome of the squared differences and multiplies them by their expected occurrence rate which is $\frac{1}{n}$. The possible outcomes will always be the same in every simulation and therefore can be treated as a constant.

The source of variation comes from how many times each outcome occurs. If $\frac{1}{n}$ is therefore factored out of (7), we only need to replace $\frac{1}{n}$ with the appropriate random variable to find the probability density function of the mean of squared differences.

Since $\frac{1}{n}$ is a proportion and the distribution of the squared differences appears to be normal (as Figure 11 illustrates), it is reasonable to assume that the random variable might be closely related to the binomial distribution with $p = \frac{1}{n}$. To test this hypothesis, we will divide the expected value of the sum of all possible squared distances out of each element represented in Figure 11 and place the new results in a histogram. By dividing by the sum of all possible squared differences, we will remove the constant part of the equation, as discussed above. The only portion of the equation left will be the source of variation, which is a proportion. The expected value of this proportion should be $\frac{1}{n}$. To see if this proportion follows the normal approximation of a binomial distribution, we will overlay a normal distribution with a mean equal to $\frac{1}{n}$ and a standard deviation equal to $\sqrt{\frac{1/n(1-1/n)}{n^2}}$. The denominator must be n^2 instead of n because each item in the vector can move to all *n* available positions in the second vector so there are $n \times n$ or n^2 possible outcomes. The code for this test can be found in section 1 of the appendix. The results are below in Figure 12.

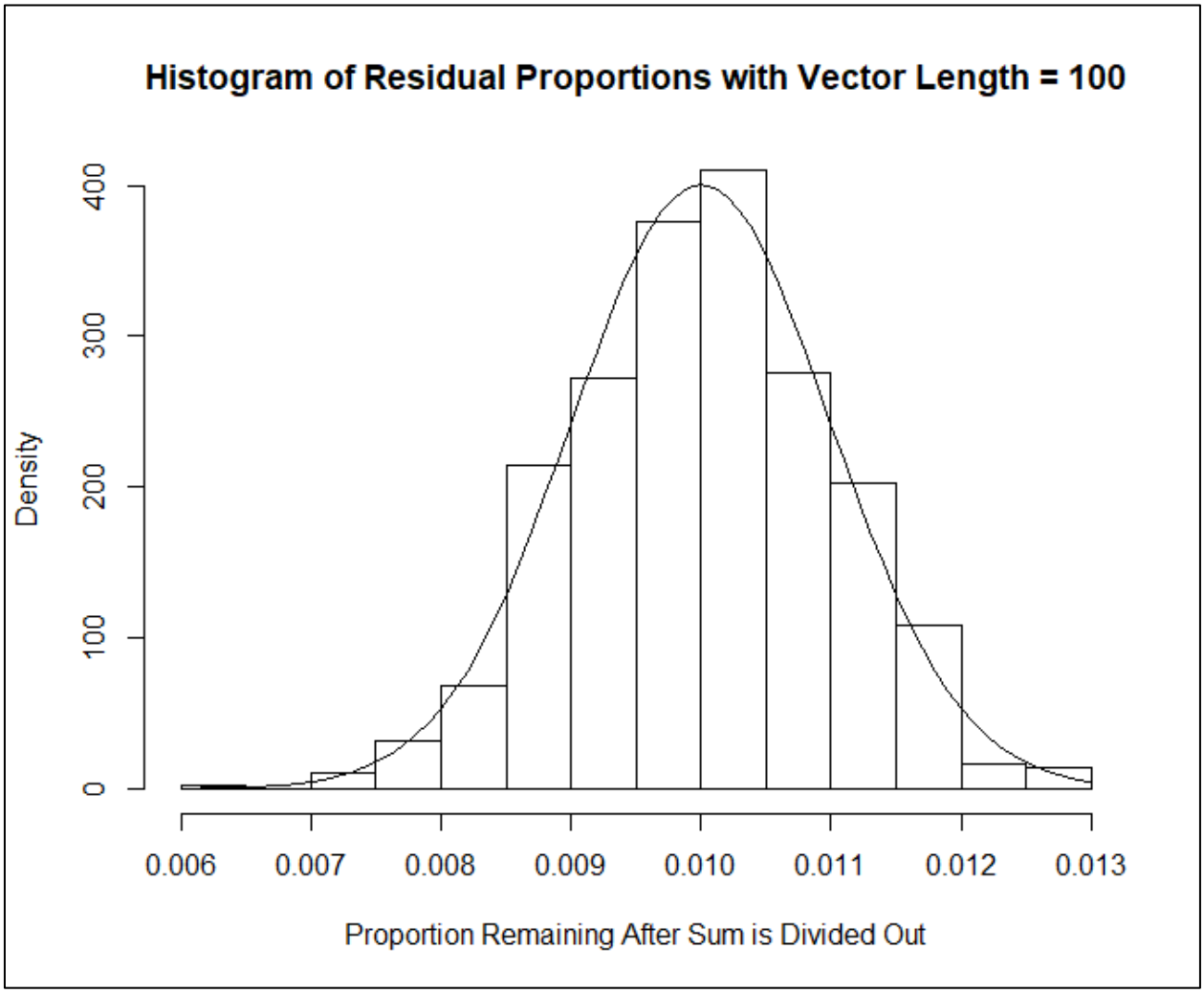


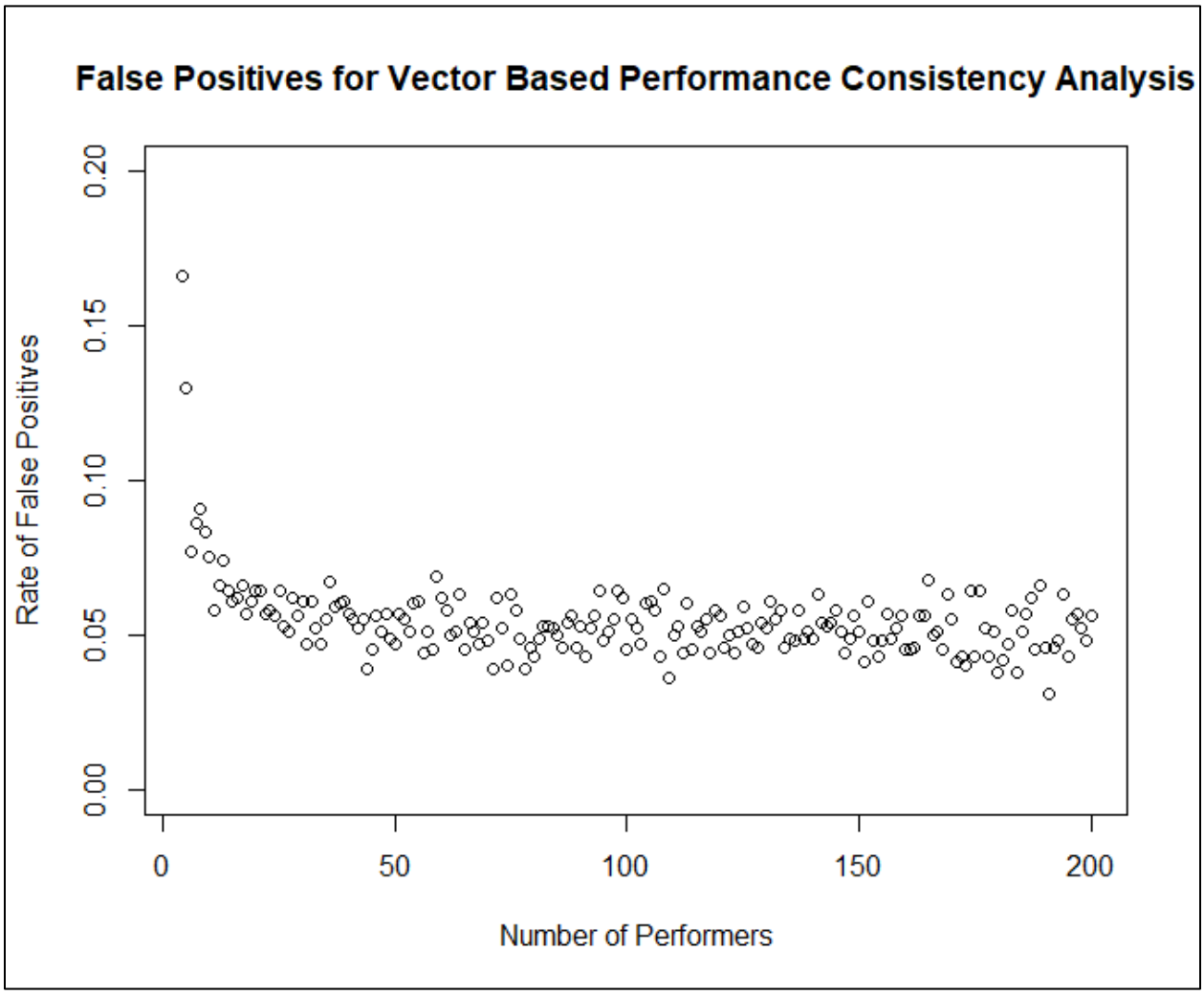
Figure 12

Figure 12 reveals that this probability density function is a good fit for the distribution observed through simulation. Multiplying this distribution by the expected mean squared difference therefore yields a comparison threshold for a hypothesis test with $\alpha = .05$. The final formula for probability density function and the formula for the comparison threshold are expressed below in (8) and (9), respectively:

$$\frac{\sum_{i=1}^{n-1} 2i(n-i)^2}{n} \times N(\mu = {}^1\!/_n, \sigma = \sqrt{\frac{{}^1\!/_n\left(1 - {}^1\!/_n\right)}{n^2}}) \quad (8)$$

$$\frac{\sum_{i=1}^{n-1} 2i(n-i)^2}{n} \times \left({}^{1}/_{n} - \left(1.645 \times \sqrt{\frac{{}^{1}/_{n}\left(1 - {}^{1}/_{n}\right)}{n^2}}\right)\right) \quad (9)$$

As with the quartile based method, we will first confirm that $\alpha = .05$ by running a simulation where the performance is completely determined by random factors and determining the rate of false positives the test yields. The test will be run 1,000 times for each *n* from 4 to 200. The code for this test can be found in Section 1 of the Appendix. The results are below in Figure 13:



Mean = .055

Figure 13

As Figure 13 illustrates, this test yields a false positive approximately 5% of the time, as expected from a test with a level of significance of 5%. Next, we will test the power of this test using the same method we used to test the power of the quartile method that is described above in Table 9 and Table 10. As was done with the quartile method, the power will be tested with a performance factor of 10% and 50%. Each test will be run 1,000 times for $n = (4, 5, 6, \ldots 100)$. The results will then be compared to the results of the power calculated for the quartile version of the Performance Consistency Analysis. The code for these tests can be found in Section 1 of the Appendix. The results are shown below in Figure 14:

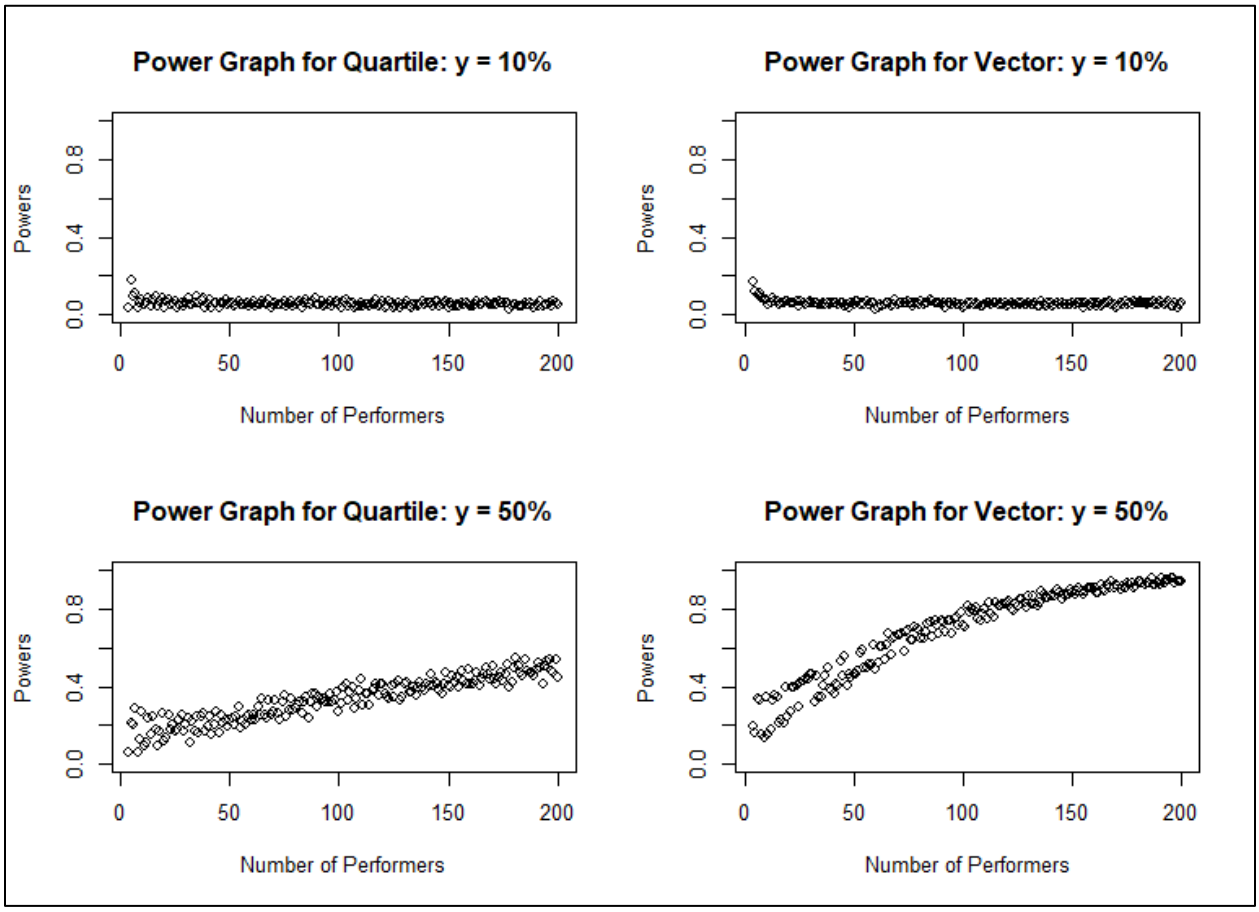


Figure 14

As Figure 14 illustrates, both the Quartile and Vector methods are very ineffective at identifying that the performer's influence is more than 0% when the performer's influence on the

outcome is 10% or less. When the performers influences 50% of the outcome, however, the vector method proves to be much more powerful than the quartile method. Specifically, the vector method appears to be approximately twice as powerful as the quartile method for any number of performers *n* when y=50%. This vector based Performance Consistency Analysis offers a useful and powerful means to evaluate how consistently performers perform from one period to the next relative to each other performer. The vector based Performance Consistency Analysis can be used to determine if a performer has some influence on the outcome of the metric. If the analysis suggests that a performer does influence the outcome, the metric may serve as a viable performance metric to evaluate performance and identify strong and weak performers.

Opportunities for Additional Research

While the versions of the methods discussed in this paper offer a viable and valuable solution to the problems they are designed to address, additional research could serve to further improve these methods. The Capacity Weighting Correlation analysis could be improved by testing the method under different conditions. For example, the analysis could be tested using p-values other than 0.05 and 0.5, using different numbers of performers and tasks, and using more than two months of data. Additionally, the analysis could best tested to determine how it handles conditions that occur in real production environments but were not included in the simulations in this paper. These conditions include non-equal probability of each task being assigned, some performers having a higher likelihood of being assigned a certain task than other performers, and some performers being more efficient at completing certain tasks than others.

The Performance Consistency Analysis could also be improved by utilizing information that was not used in the vector based version of the method. For example, the Performance Consistency Analysis utilized the full rankings of performance but did not utilize the variation of the actual performance scores. Additional, multiple periods of data could be used to improve the power of the test. The analysis could also be further evaluated by testing the analysis under different levels of *y* (the degree to which performers influence the outcome of the metric) and under different methods for simulating the random and non-random effects on the performance. Finally, the tests usefulness could be improved by establishing standards for accepting or rejecting a potential performance metric. While these opportunities could lead to improvements for these methods, the current analyses are sufficiently powerful to offer viable solutions to real problems in today's complex business environment.

Conclusion

The Performance Consistency Analysis and the Capacity Weighting Correlation Analysis are both viable analytical methods that offer a valuable solution to common problems encountered when measuring productivity and performance. Developing these methods involved creating hypotheses about how performance data should logically behave, expressing these behaviors numerically, testing these behaviors to ensure they hold consistently, then leveraging these behaviors to develop powerful methods to estimate task weightings and how consistently employees perform. This approach led to the Capacity Weighting Correlation Analysis which exploits the relationship between the number of each task performed and the overall weighted productivity score to identify potentially under and over-weighted tasks. The method then adjusts the weights until no under or over-weighted tasks can be identified. The vector based Capacity Weighting Correlation analysis uses performance rankings over time to measure

performance consistency and evaluate to what extent the performance is influenced by external factors. The enhancements made to these methods through this research resulted in powerful analyses that offer real and viable solutions to significant problems that many businesses encounter. Used appropriately, these methods could serve to accurately and inexpensively identify and improve various types of performance metrics including productivity, turnaround time, and quality. Considering how fundamental these types of metrics are to any operating environment, these methods offer a valuable asset to any organization striving towards operational excellence.

Works Cited

James B. Schreiber. 2016. *Motivation 101.* Springer Publishing Company.

Kathryn E Merrick, Kamran Shafi. 2011. "Achievement, affiliation, and power: Motive profiles for artificial agents." *Adaptive Behavior* 19 (1): 40-62.

Machol, Robert E. 1975. "Principles of Operations Research." *Interfaces* 5, no. 2 (Part 1 of 2, February): 31-32. http://www.jstor.org/stable/25059155.

Nicewander, Joseph Lee Rogers and W. Alan. 1988. "Thirteen Ways to Look at the Correlation Coefficient." *The American Statistician* 42, no. 1 (February): 59-66.

Appendix

Section 1

Performance Consistency Analysis Code

```
rm(list = ls()) #reset the workspace


*Functions to be used in analysis*

assignQuart<-function(mth,q1,q2,q3){ #this function takes a vector of performance and their quartiles and creates a vector of the quartiles that each performer falls into

result<-NULL

for (i in 1:length(mth)) {

if(mth[i]<q1)

thisResult<-1

else if(mth[i]<q2)

thisResult<-2

else if(mth[i]<q3)

thisResult<-3

else

thisResult<-4

result<-c(result,thisResult)

}

return(result)

}


checkSame<-function(m1,m2) { #This function takes the quartile results from two months and determines how many of the performers were the same by returning a vector of values "2" for same and "1" for different

results.vec<-NULL

for (i in 1:length(m1)) {

if (m1[i]==m2[i])
```

```
thisResult<-2

else

thisResult<-1

results.vec<-c(results.vec,thisResult)

}

return(results.vec)

}
```

```
randTest2<-function(p){ #This two month test generates an instance of the results when per=0 (performers do not influence outcome), given p performers. It returns a percent of performers who acheived the same performance both months.  This function will be used to calculate conf. intervals.

Onemth.vec<-NULL #create vectors to store results

Twomth.vec<-NULL


month1<-runif(p,0,1) #generate a vector of p performances for both months

month2<-runif(p,0,1)


m1q1<-quantile(month1,0.25) #now we calculate the quantiles

m2q1<-quantile(month2,0.25)


m1q2<-quantile(month1,0.5)

m2q2<-quantile(month2,0.5)


m1q3<-quantile(month1,0.75)

m2q3<-quantile(month2,0.75)


month1Results<-assignQuart(month1,m1q1,m1q2,m1q3) #now we use the assignQuart)function to obtain a vector with quartile performances

month2Results<-assignQuart(month2,m2q1,m2q2,m2q3)
```

```
answer<-checkSame(month1Results,month2Results) #now we use the checkSame() function to obtain a vector indicating how many results were the same


success<-length(which(answer==2)) #count how many performers acheived the same results


return(success/p) #return the answer!

}


*Functions to test theoretical threshold for testing*

simResults<-function(pa){ #create a function that generates a vector of random results given some number of performers (represented by pa)

randResults.vec<-NULL #create a vector to store results

for (i in 1:1000) {

answ<-randTest2(pa)

randResults.vec<-c(randResults.vec,answ)

}

return(randResults.vec)

}


simResultsVec<-function(numb){ #create a function that generates the 95th percentile for 4 to numb performers using the simResults function

ansVec<-NULL #create a vector to store the results

for (i in 4:numb){

tempVec<-simResults(i)

ansVec<-c(ansVec,quantile(tempVec,.95)) #add the 95th percentile to ansVec

}

return(ansVec)

}
```

```
calcResultsVec<-function(numbe){ #create a function that generates the 95th percentile for 4 to numbe performers using the simResults function

ansVec<-NULL #create a vector to store the results

for (i in 4:numbe){

tempAns<-(.25+(1.645*sqrt(1/i*.25*.75))) #calculate the theoretical 95th percentile

ansVec<-c(ansVec,tempAns) #add the 95th percentile to ansVec

}

return(ansVec)

}


firstVec<-simResultsVec(200)

secVec<-calcResultsVec(200)

difVec<-firstVec-secVec #calculate the differences between the simulated and theoretical results


p<-seq(4,200,by=1)#create a vector to plot the results over

plot.new()

plot(p,difVec, main="Difference Between Calculated and Simulated 95th Percentile",xlab="Number of Performers",ylab="Difference",xlim=c(0,200),ylim=c(-.2,0.2)) #plot the differences against p from 4 to 200


rejectNullTest<-function(p){ #create a function that returns the rate of rejection of the null hypothesis given the number of performers p

successes<-0 #create an integer to count successes

for (i in 1:1000){ #run a for loop p times

thisTry<-randTest2(p) #generate a result using the randTest2()

if (thisTry>=(.25+(1.645*sqrt(1/p*.25*.75)))) #compare to the theoretical mean

successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

return(successes/1000) #return the proportion of 1000 trials that were successful
```

```
}

rejectNullVec<-function(number){ #create a function that tests for false positives from p equals 25 to number

ansVec<-NULL #create a vector to store the results

for (i in 25:number){

tempAns<-rejectNullTest(i)

ansVec<-c(ansVec,tempAns) #add the most recent results to the answer vector

}

return(ansVec)

}


falsNegVec<-rejectNullVec(200)

p<-seq(25,200,by=1)#create a vector to plot the reults over

plot.new()

plot(p,falsNegVec,main="False Positives",xlab="Number of Performers",ylab="Rate of False Positive",xlim=c(25,200),ylim=c(0,0.2)) #plot the proportion of false positives from 4 to 200

mean(falsNegVec)
```

Calculating the Powers

```
perTestNorm<-function(p,per){ #This two month test generates an instance of the results given per (performers have per% influence over the outcome) and given p performers. It returns a percent of performers who acheived the same performance both months.  This function assumes that per is normally distributed.

month1<-NULL #create vectors to store results

month2<-NULL


invCDFset <- 1/p #create a variable that represents a probability that we will use to calculate an inverse CDF in the next line

for (i in 1:p){ #create a for loop that generates the performance of p performers
```

```
perContr<-qnorm(invCDFset,mean=50,sd=50/3) #create a variable that represents the contribution by performance.  Use cdf of normal curve where 99.7% of the data falls within 0 to 100 which is th range of outCon

if(perContr < 0){

perContr = 0 #if perContr ended up being less than 0, then set it to 0

}

outContr<-runif(1,0,100) #generate a performance contribution from outside factors

thisPerf<-(((per/100)*perContr)+((1-(per/100))*outContr)) #calculate this performance as a combo of perContr and outContr

month1<-c(month1,thisPerf) #add this performance

if(invCDFset + (1/p) < 1){

invCDFset<-invCDFset + (1/p) #increment invCDFset by 1/p so the cdf for perContr yields a higher number for the next performer

}

}


invCDFset <- 1/p #rerun all that code for month 2

for (i in 1:p){ #create a for loop that generates the performance of p performers

perContr<-qnorm(invCDFset,mean=50,sd=50/3) #create a variable that represents the contribution by performance.  Use cdf of normal curve where 99.7% of the data falls within 0 to 100 which is th range of outCon

if(perContr < 0){

perContr = 0 #if perContr ended up being less than 0, then set it to 0

}

outContr<-runif(1,0,100) #generate a performance contribution from outside factors

thisPerf<-(((per/100)*perContr)+((1-(per/100))*outContr)) #calculate this performance as a combo of perContr and outContr

month2<-c(month2,thisPerf) #add this performance

if(invCDFset + (1/p) < 1){

invCDFset<-invCDFset + (1/p) #increment invCDFset by 1/p so the cdf for perContr yields a higher number for the next performer, unless this would bring the value to one

}
```

```
}



m1q1<-quantile(month1,0.25) #now we calculate the quantiles

m2q1<-quantile(month2,0.25)


m1q2<-quantile(month1,0.5)

m2q2<-quantile(month2,0.5)


m1q3<-quantile(month1,0.75)

m2q3<-quantile(month2,0.75)


month1Results<-assignQuart(month1,m1q1,m1q2,m1q3) #now we use the assignQuart)function to obtain a vector with quartile performances

month2Results<-assignQuart(month2,m2q1,m2q2,m2q3)


answer<-checkSame(month1Results,month2Results) #now we use the checkSame() function to obtain a vector indicating how many results were the same


success<-length(which(answer==2)) #count how many performers acheived the same results


return(success/p) #return the answer!

}


powerCalcp10<-function(numbers){ #create a function that calculates the power given p from 4 to 200 that will use 'numbers' simulations

success<-NULL #create a vector to store results

for (i in 4:200){ #run a loop for each value of p

successes=0 #create an int variable to count results

for (j in 1:numbers){ #run an inner loop 1000 times to calculate success rate
```

```
thisTry<-perTestNorm(i,10) #generate a result using the perTest2function

if (thisTry>=(.25+(1.645*sqrt(1/i*.25*.75)))) #calculate success using theoretical threshold

successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

success<-c(success,successes/numbers)

}

return(success)

}


powers<-powerCalcp10(1000)#run the function and store the results




p<-seq(4,200,by=1)#create a vector to plot the results over

plot.new()

plot(p,powers,main="Powers with y=10%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,0.2))

mean(powers)



***Now run the results again with per=50***

powerCalcp50<-function(numbers){ #create a function that calculates the power given p from 4 to 200 that will use 'numbers' simulations

success<-NULL #create a vector to store results

for (i in 4:200){ #run a loop for each value of p

successes=0 #create an int variable to count results

for (j in 1:numbers){ #run an inner loop 1000 times to calculate success rate

thisTry<-perTestNorm(i,50) #generate a result using the perTest2function

if (thisTry>=(.25+(1.645*sqrt(1/i*.25*.75)))) #calculate success using theoretical threshold
```

```
successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

success<-c(success,successes/numbers)

}

return(success)

}


powers<-powerCalcp50(1000)#run the function and store the results



p<-seq(4,200,by=1)#create a vector to plot the results over

plot.new()

plot(p,powers,main="Powers with y=50%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,0.6))

mean(powers)



**Vector mean test**

vecDifMeanCalc<-function(vecLength){ #this formula calculates the expected mean of the squared differences between a cardinally ordered vector and a randomly ordered vector given the vector length

ans<-0 #initiate the answer to 0

for(i  in 1:(vecLength-1)){

thisTime<-2*(vecLength-i)^2*i

ans<-ans+thisTime

}

return(ans/vecLength^2)

}
```

```
vecMeanCalcTest<-function(n){ #this function tests the vecDifMeanCalc function by comparing it to simulated results using a max vector length n and returns a vector of the differences from 4 to n

resultsVec<-NULL #create a vector to store the results from the simulation

for(i in 4:n){ #run a loop for each length of n

randomVec<-NULL #create a vector to store the results of the 1,000 random tests

for (j in 1:10000) { #run a loop 10,000 times to get 10,000 results

scores<-runif(i,1,100) #produce random scores

ranks<-rank(scores) #calculate ranks

orig<-seq(1,i,by=1) #create the original ranking vector

randomVec<-c(randomVec,mean((ranks-orig)^2)) #add the mean of the squared differences to the resultsVec

}

resultsVec<-c(resultsVec,mean(randomVec)-vecDifMeanCalc(i)) #add the difference of the calculated and simulated result to the results vector

}

return(resultsVec)

}


differences<-vecMeanCalcTest(200) #run the function for n=200

vectorLength<-seq(4,200,by=1)#create a vector to plot the results over

plot.new()

plot(vectorLength,differences,main="Difference Between Calculated and Simulated Means",xlab="Number of Performers",ylab="Differences",xlim=c(4,200),ylim=c(-30,30)) #plot the differences

mean(abs(differences))


vecDifPropCalc<-function(vecLength){ #this formula modified vecDifMeanCalc by factoring out 1/n to allow this fraction to be replaced by a random variable

ans<-0 #initiate the answer to 0

for(i  in 1:(vecLength-1)){

thisTime<-2*(vecLength-i)^2*i
```

```
ans<-ans+thisTime
}
return(ans/vecLength)
}
```

Determining Standard Deviation*

```
genDifVals<-function(len){ #this function will be used to generate 1,000 values of the mean of squared differences using random simulation
randomVec<-NULL #create a vector to store the results of the 1,000 random tests
for (j in 1:1000) { #run a loop 1,000 rimtes to get 1,000 results
scores<-runif(len,1,100) #produce random scores
ranks<-rank(scores) #calculate ranks
orig<-seq(1,len,by=1) #create the original ranking vector
randomVec<-c(randomVec,mean((ranks-orig)^2)) #add the mean of the squared differences to the resultsVec
}
return(randomVec)
}

vecLength100sqDif<-genDifVals(100)

hist(vecLength100sqDif, main="Histogram of Mean of Squared Difference with Vector Length = 100",xlab="Mean of Squared Difference")

normVec<-vecLength100sqDif/vecDifPropCalc(100)

hist(normVec, main="Histogram of Residual Proportions with Vector Length = 100", freq=FALSE, xlab="Proportion Remaining After Sum is Divided Out")
xaxis<-seq(.006,.013,by=.0001)
```

```
lines(xaxis,dnorm(xaxis,mean=1/100,sd=sqrt(1/100*(1-(1/100))/100^2)) )

###^^^This seems to work!
```

Testing the Vector Based Performance Consistency Analysis for False Positives

```
randTest2vec<-function(p){ #This two month test generates an instance of the results when per=0 (performers do not influence outcome), given p performers. It returns the mean of squared differences using the vecotr approach.

Onemth.vec<-NULL #create vectors to store results

Twomth.vec<-NULL


month1<-runif(p,0,1) #generate a vector of p performances for both months

month2<-runif(p,0,1)


m1Vec<-rank(month1) #calculate the rankings for each month

m2Vec<-rank(month2)


ans<-mean((m1Vec-m2Vec)^2) #calculate the mean of the squared differences


return(ans) #return the answer!

}


simResultsVec<-function(pa){ #create a function that generates a vector of random results given some number of performers (represented by pa)

randResults.vec<-NULL #create a vector to store results

for (i in 1:1000) {

answ<-randTest2vec(pa)

randResults.vec<-c(randResults.vec,answ)

}

return(randResults.vec)

}
```

```
rejectNullTestVec<-function(p){ #create a function that returns the rate of rejection of the null hypothesis given the number of performers p

successes<-0 #create an integer to count successes

for (i in 1:1000){ #run a for loop 1000 times

thisTry<-randTest2vec(p) #generate a result using the randTest2vec()

if (thisTry<=(vecDifPropCalc(p)*qnorm(.05,mean=1/p,sd=sqrt(1/p*(1-(1/p))/p^2)))) #compare to the theoretical mean

successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

return(successes/1000) #return the proportion of n trials that were successful

}


rejectNullVec<-function(number){ #create a function that tests for false negatives from p equals 25 to number

ansVec<-NULL #create a vector to store the results

for (i in 4:number){

tempAns<-rejectNullTestVec(i)

ansVec<-c(ansVec,tempAns) #add the most recent results to the answer vector

}

return(ansVec)

}


falsNegVec<-rejectNullVec(200)

p<-seq(4,200,by=1)

plot(p,falsNegVec,main="False Positives for Vector Based Performance Consistency Analysis",xlab="Number of Performers",ylab="Rate of False Positives",xlim=c(4,200),ylim=c(0,0.2))

mean(falsNegVec)
```

Compare the Powers of the Quartile Method and the Vector Method

par(mfrow=c(2,2))

perTestNormvec<-function(p,per){ #This two month test generates an instance of the results given per (performers have per% influence over the outcome) and given p performers. It returns a percent of performers who acheived the same performance both months. This function assumes that per is normally distributed.

month1<-NULL #create vectors to store results

month2<-NULL

invCDFset <- 1/p #create a variable that represents a probability that we will use to calculate an inverse CDF in the next line

for (i in 1:p){ #create a for loop that generates the performance of p performers

perContr<-qnorm(invCDFset,mean=50,sd=50/3) #create a variable that represents the contribution by performance. Use cdf of normal curve where 99.7% of the data falls within 0 to 100 which is th range of outCon

if(perContr < 0){

perContr = 0 #if perContr ended up being less than 0, then set it to 0

}

outContr<-runif(1,0,100) #generate a performance contribution from outside factors

thisPerf<-(((per/100)*perContr)+((1-(per/100))*outContr)) #calculate this performance as a combo of perContr and outContr

month1<-c(month1,thisPerf) #add this performance

if(invCDFset + (1/p) < 1){

invCDFset<-invCDFset + (1/p) #increment invCDFset by 1/p so the cdf for perContr yields a higher number for the next performer

}

}

invCDFset <- 1/p #rerun all that code for month 2

for (i in 1:p){ #create a for loop that generates the performance of p performers

```
perContr<-qnorm(invCDFset,mean=50,sd=50/3) #create a variable that represents the contribution by performance.  Use cdf of normal curve where 99.7% of the data falls within 0 to 100 which is th range of outCon

if(perContr < 0){

perContr = 0 #if perContr ended up being less than 0, then set it to 0

}

outContr<-runif(1,0,100) #generate a performance contribution from outside factors

thisPerf<-(((per/100)*perContr)+((1-(per/100))*outContr)) #calculate this performance as a combo of perContr and outContr

month2<-c(month2,thisPerf) #add this performance

if(invCDFset + (1/p) < 1){

invCDFset<-invCDFset + (1/p) #increment invCDFset by 1/p so the cdf for perContr yields a higher number for the next performer, unless this would bring the value to one

}

}



m1Vec<-rank(month1) #calculate the rankings for each month

m2Vec<-rank(month2)


ans<-mean((m1Vec-m2Vec)^2) #calculate the mean of the squared differences


return(ans) #return the answer!

}



powerCalcp10vec<-function(numbers){ #create a function that calculates the power given p from 4 to 200 that will use 'numbers' simulations

success<-NULL #create a vector to store results

for (i in 4:200){ #run a loop for each vector length
```

```
successes=0 #create an int variable to count results

for (j in 1:numbers){ #run an inner loop 1000 times to calculate success rate

thisTry<-perTestNormvec(i,10) #generate a result using the perTest2function

if (thisTry<=(vecDifPropCalc(i)*qnorm(.05,mean=1/i,sd=sqrt(1/i*(1-(1/i))/i^2)))) #compare to the theoretical mean

successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

success<-c(success,successes/numbers)

}

return(success)

}


powerCalcp50vec<-function(numbers){ #create a function that calculates the power given p from 4 to 200 that will use 'numbers' simulations

success<-NULL #create a vector to store results

for (i in 4:200){ #run a loop for each vector length

successes=0 #create an int variable to count results

for (j in 1:numbers){ #run an inner loop 1000 times to calculate success rate

thisTry<-perTestNormvec(i,50) #generate a result using the perTest2function

if (thisTry<=(vecDifPropCalc(i)*qnorm(.05,mean=1/i,sd=sqrt(1/i*(1-(1/i))/i^2)))) #compare to the theoretical mean

successes<-successes+1 #if the results exceeded the 95% confidence interval, increment successes else do nothing

}

success<-c(success,successes/numbers)

}

return(success)

}
```

powers10qt<-powerCalcp10(1000)#run the function and store the results

powers10vec<-powerCalcp10vec(1000)#run the function and store the results

powers50qt<-powerCalcp50(1000)#run the function and store the results

powers50vec<-powerCalcp50vec(1000)#run the function and store the results

n<-seq(4,200,by=1)

plot(n,powers10qt, main = "Power Graph for Quartile: y = 10%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,1))

plot(n,powers10vec, main = "Power Graph for Vector: y = 10%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,1))

plot(n,powers50qt, main = "Power Graph for Quartile: y = 50%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,1))

plot(n,powers50vec, main = "Power Graph for Vector: y = 50%",xlab="Number of Performers",ylab="Powers",xlim=c(4,200),ylim=c(0,1))

Section 2

Capacity Weighting Correlation Analysis

rm(list = ls()) #reset the workspace

Data to test the correlation concept

UDcorrTest<-function(n,w1,w2,w3,s1,s2,s3){ #This function will produce scores and proportions for n performers with work items with actual weights w1, w2, and w3 and starting weights s1, s2, and s3

prime=0

ansMat<-matrix(nrow=4,ncol=n) #create a matrix to store the answer with a row for the total weighted score, a row for each proportion, and a column for each performer

for(i in 1:n) { #run a loop for each performer to assign credit and calculate scores and proportions

capacity = 100 #set capacity to 100

score = 0 + prime #set score

WI1<-0 #initiate work item 1 count to 0

WI2<-0 #initiate work item 2 count to 0

```
WI3<-0 #initiate work item 3 count to 0
while(capacity>0){ #run a while loop while there is still capacity left
thisWI<-floor(runif(1,1,4)) #generate a random integer between 1 and 3
if (thisWI==1) {
capUse <- w1
thisScore = s1
WI1<-WI1+1
} else if (thisWI==2) {
capUse <- w2
thisScore = s2
WI2<-WI2+1
} else {
capUse <- w3
thisScore = s3
WI3<-WI3+1
}
capacity = capacity - capUse #remove the capacity associated with this work item
score <- score + thisScore
if(capacity<0){ #if you have run over capacity, reverse the WI and score increment
score = score-thisScore
if (thisWI==1) {
WI1<-WI1-1
} else if (thisWI==2) {
WI2<-WI2-1
} else {
WI3<-WI3-1
}
}
```

```
}
ansMat[,i]=c(score,WI1,WI2,WI3) #update the ith column with the score and the three work item counts
prime<-prime+.0001
} #close the for loop
return (ansMat)
} #close the function



corrTestResults<-function(num,per,we1,we2,we3) {#This function will run the UDcorrTest funtion num times with per performers and will return the correlation coefficients and p-values for all num simulations
ansMat<-matrix(nrow=6,ncol=num) #create a matrix to store the answer with a row for each correlation coefficient and a row for each p-value

for(i in 1:num){
thisSim<-UDcorrTest(per,we1,we2,we3,1,1,1)

cor1<-cor(thisSim[1,],thisSim[2,]) # calculate the correlations and p values
test1<-cor.test(thisSim[1,],thisSim[2,])
p1<-test1$p.value

cor2<-cor(thisSim[1,],thisSim[3,])
test2<-cor.test(thisSim[1,],thisSim[3,])
p2<-test2$p.value

cor3<-cor(thisSim[1,],thisSim[4,])
test3<-cor.test(thisSim[1,],thisSim[4,])
p3<-test3$p.value
```

```
ansMat[1,i]=cor1
ansMat[2,i]=cor2
ansMat[3,i]=cor3

ansMat[4,i]=p1
ansMat[5,i]=p2
ansMat[6,i]=p3
}

return(ansMat)
}

*Run the functions and read out the results*
thisCorTest<-corrTestResults(1000,100,1,5,10)

mean(thisCorTest[1,])
mean(thisCorTest[2,])
mean(thisCorTest[3,])
mean(thisCorTest[4,])
mean(thisCorTest[5,])
mean(thisCorTest[6,])

quantile(thisCorTest[1,],.05)
quantile(thisCorTest[2,],.05)
quantile(thisCorTest[3,],.05)

quantile(thisCorTest[1,],.95)
quantile(thisCorTest[2,],.95)
```

```
quantile(thisCorTest[3,],.95)

thisCorTest<-corrTestResults(1000,100,1,1,1)

mean(thisCorTest[1,])
mean(thisCorTest[2,])
mean(thisCorTest[3,])
mean(thisCorTest[4,])
mean(thisCorTest[5,])
mean(thisCorTest[6,])

quantile(thisCorTest[1,],.05)
quantile(thisCorTest[2,],.05)
quantile(thisCorTest[3,],.05)

quantile(thisCorTest[1,],.95)
quantile(thisCorTest[2,],.95)
quantile(thisCorTest[3,],.95)
```

Create Functions to Run Analysis with Equal Productivity

```
perfDataGen<-function(n,w1,w2,w3){ #This function will produce production data for n peformers with work items with actual weights w1, w2, and w3
prime=0
ansMat<-matrix(nrow=3,ncol=n) #create a matrix to store a column of information for each performer with the number of each of the three work items produced
for(i in 1:n) { #run a loop for each performer to assign credit and calculate scores and proportions
capacity = 100 #set capacity to 100
score = 0 + prime #set score
```

```
WI1<-0 #initiate work item 1 count to 0
WI2<-0 #initiate work item 2 count to 0
WI3<-0 #initiate work item 3 count to 0
while(capacity>0){ #run a while loop while there is still capacity left
thisWI<-floor(runif(1,1,4)) #generate a random integer between 1 and 3
if (thisWI==1) { #evaluate which item was randomly produced and increment that WI count
capUse <- w1
WI1<-WI1+1
} else if (thisWI==2) {
capUse <- w2
WI2<-WI2+1
} else {
capUse <- w3
WI3<-WI3+1
}
capacity = capacity - capUse #remove the capacity associated with this work item

if(capacity<0){ #if you have run over capacity, reverse the WI and score increment
if (thisWI==1) {
WI1<-WI1-1
} else if (thisWI==2) {
WI2<-WI2-1
} else {
WI3<-WI3-1
}
}

}
ansMat[,i]=c(WI1,WI2,WI3) #update the ith row with the three score counts
```

```
prime<-prime+.0001
} #close the for loop
return (ansMat)
} #close the function
```

******

```
wtdScoreCalc<-function(n,mat,s1,s2,s3){ #This function will produce final scores n peformers given weights s1, s2, and s3 and a 3 by n matrix containing counts of each work item produced
ansMat<-NULL #create an empty vector to store the results
for(i in 1:n){ #run a loop through each performer
thisScore<-(s1*mat[1,i])+(s2*mat[2,i])+(s3*mat[3,i])
ansMat<-c(ansMat,thisScore)
} #close the for loop
return (ansMat)
} #close the function
```

******

```
findWeightsProdSame<-function(n,w1,w2,w3){ #create a function that finds the weights given n performers and actual weights of w1, w2, and w3.  The goal is to get w1, w2, and w3.  Assumes same productivity across all performers.
wght1<-1 #initialize the weights to 1
wght2<-1
wght3<-1
counter <-0 #create a counter so the while loop doesn't run forever
results<-perfDataGen(n,w1,w2,w3)
while(TRUE){ #run a while loop to refine the weightings
scores<-wtdScoreCalc(n,results,wght1,wght2,wght3) #generate the weighted scores
```

```
cor1<-cor(scores,results[1,]) # calculate the correlations and p values

test1<-cor.test(scores,results[1,])

p1<-test1$p.value


cor2<-cor(scores,results[2,])

test2<-cor.test(scores,results[2,])

p2<-test2$p.value


cor3<-cor(scores,results[3,])

test3<-cor.test(scores,results[3,])

p3<-test3$p.value


corSum<-0 #create a variable for the sum of all the p's

corCount<-0 #create a variable for the count of all the p's


p1bool<-FALSE #create boolean values to indicates whether or not the p-values are less than .05

p2bool<-FALSE

p3bool<-FALSE


if(p1<.05){ #if the p-value of the correlation coefficient for wi1 is less than .05 then add it to the sum and
trigger the pvalue boolean variable

corSum<-corSum+cor1

corCount<-corCount+1

p1bool<-TRUE

}


if(p2<.05){ #if the p-value of the correlation coefficient for wi2 is less than .05 then add it to the sum and
trigger the pvalue boolean variable

corSum<-corSum+cor2
```

```
corCount<-corCount+1

p2bool<-TRUE

}


if(p3<.05){ #if the p-value of the correlation coefficient for wi3 is less than .05 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor3

corCount<-corCount+1

p3bool<-TRUE

}


corAvg<-corSum/corCount #calculate the average correlation coefficient using the coefficients that had p-values less than .5


if(p1bool){ #if the p-value for cor1 was less than .05, change weighting1

if(cor1<corAvg){

wght1<-wght1*1.01

}

else {wght1<-wght1*.99}

}



if(p2bool){ #if the p-value for cor2 was less than .05, change weighting2

if(cor2<corAvg){

wght2<-wght2*1.01

}

else {wght2<-wght2*.99}

}
```

```
if(p3bool){ #if the p-value for cor3 was less than .05, change weighting3
if(cor3<corAvg){
wght3<-wght3*1.01
}
else {wght3<-wght3*.99}
}

counter<-counter+1

if( ((!p1bool & !p2bool) & !p3bool) | (counter==10000) ){
break
}

}
minW<-min(wght1,wght2,wght3) #get the minimum weight to index the weights to 1
return(c(wght1/minW,wght2/minW,wght3/minW))
}



runFWtestProdS1510<-function(n) { #this function runs findWeightsprodSame() n times with weights 1, 5, and 10.  Uses 50 performers
ansMat<-matrix(nrow=3,ncol=n)
for(i in 1:n){
thisAns<-findWeightsProdSame(50,1,5,10)
ansMat[1,i]<-thisAns[1]
ansMat[2,i]<-thisAns[2]
ansMat[3,i]<-thisAns[3]
}
```

```
return(ansMat)
}


runFWtestResultsProdS1510<-runFWtestProdS1510(100)

mean(runFWtestResultsProdS1510[1,])
mean(runFWtestResultsProdS1510[2,])
mean(runFWtestResultsProdS1510[3,])

quantile(runFWtestResultsProdS1510[1,],.05)
quantile(runFWtestResultsProdS1510[2,],.05)
quantile(runFWtestResultsProdS1510[3,],.05)

quantile(runFWtestResultsProdS1510[1,],.95)
quantile(runFWtestResultsProdS1510[2,],.95)
quantile(runFWtestResultsProdS1510[3,],.95)

mean(abs(runFWtestResultsProdS1510[1,]-1))/1
mean(abs(runFWtestResultsProdS1510[2,]-5))/5
mean(abs(runFWtestResultsProdS1510[3,]-10))/10


par(mfrow=c(1,3))
hist(runFWtestResultsProdS1510[1,]-1, main = "Error Distribution for Task with Weight=1", xlab="Error for Weight 1",xlim=c(0,.3),xaxt='n')
axis(side=1, at=seq(0,0.3,by=.05), labels=paste(seq(0,30,by=5),'%'))
hist((runFWtestResultsProdS1510[2,]-5)/5, main = "Error Distribution for Task with Weight=5", xlab="Error for Weight 5",xlim=c(-.3,.3),xaxt='n')
```

```
axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))

hist((runFWtestResultsProdS1510[3,]-10)/10, main = "Error Distribution for Task with Weight=10",
xlab="Error for Weight 10",xlim=c(-.3,.3),xaxt='n')

axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))
```

Code for Analysis using p=0.5 instead of p=0.05

```
findWeightsProdSameHighp<-function(n,w1,w2,w3){ #create a function that finds the weights given n performers and actual weights of w1, w2, and w3.  The goal is to get w1, w2, and w3.  Assumes same productivity across all performers.

wght1<-1 #initialize the weights to 1

wght2<-1

wght3<-1

counter <-0 #create a counter so the while loop doesn't run forever

results<-perfDataGen(n,w1,w2,w3)

while(TRUE){ #run a while loop to refine the weightings

scores<-wtdScoreCalc(n,results,wght1,wght2,wght3) #generate the weighted scores

cor1<-cor(scores,results[1,]) # calculate the correlations and p values

test1<-cor.test(scores,results[1,])

p1<-test1$p.value


cor2<-cor(scores,results[2,])

test2<-cor.test(scores,results[2,])

p2<-test2$p.value


cor3<-cor(scores,results[3,])

test3<-cor.test(scores,results[3,])

p3<-test3$p.value


corSum<-0 #create a variable for the sum of all the p's
```

```
corCount<-0 #create a variable for the count of all the p's


p1bool<-FALSE #create boolean values to indicates whether or not the p-values are less than .5

p2bool<-FALSE

p3bool<-FALSE


if(p1<.5){ #if the p-value of the correlation coefficient for wi1 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor1

corCount<-corCount+1

p1bool<-TRUE

}


if(p2<.5){ #if the p-value of the correlation coefficient for wi2 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor2

corCount<-corCount+1

p2bool<-TRUE

}


if(p3<.5){ #if the p-value of the correlation coefficient for wi3 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor3

corCount<-corCount+1

p3bool<-TRUE

}


corAvg<-corSum/corCount #calculate the average correlation coefficient using the coefficients that had p-values less than .5
```

```
if(p1bool){ #if the p-value for cor1 was less than .5, change weighting1
if(cor1<corAvg){
wght1<-wght1*1.01
}
else {wght1<-wght1*.99}
}

if(p2bool){ #if the p-value for cor2 was less than .5, change weighting2
if(cor2<corAvg){
wght2<-wght2*1.01
}
else {wght2<-wght2*.99}
}

if(p3bool){ #if the p-value for cor3 was less than .5, change weighting3
if(cor3<corAvg){
wght3<-wght3*1.01
}
else {wght3<-wght3*.99}
}

counter<-counter+1

if( ((!p1bool & !p2bool) & !p3bool) | (counter==10000) ){
break
}
```

```
}
minW<-min(wght1,wght2,wght3) #get the minimum weight to index the weights to 1
return(c(wght1/minW,wght2/minW,wght3/minW))
}


runFWtestProdSHighp1510<-function(n) { #this function runs findWeightsprodSame() n times with weights 1, 5, and 10.  Uses 50 performers
ansMat<-matrix(nrow=3,ncol=n)
for(i in 1:n){
thisAns<-findWeightsProdSameHighp(50,1,5,10)
ansMat[1,i]<-thisAns[1]
ansMat[2,i]<-thisAns[2]
ansMat[3,i]<-thisAns[3]
}
return(ansMat)
}


runFWtestResultsProdSHighp1510<-runFWtestProdSHighp1510(100)

mean(runFWtestResultsProdSHighp1510[1,])
mean(runFWtestResultsProdSHighp1510[2,])
mean(runFWtestResultsProdSHighp1510[3,])

quantile(runFWtestResultsProdSHighp1510[1,],.05)
quantile(runFWtestResultsProdSHighp1510[2,],.05)
quantile(runFWtestResultsProdSHighp1510[3,],.05)
```

```
quantile(runFWtestResultsProdSHighp1510[1,],.95)

quantile(runFWtestResultsProdSHighp1510[2,],.95)

quantile(runFWtestResultsProdSHighp1510[3,],.95)


mean(abs(runFWtestResultsProdSHighp1510[1,]-1))/1

mean(abs(runFWtestResultsProdSHighp1510[2,]-5))/5

mean(abs(runFWtestResultsProdSHighp1510[3,]-10))/10



par(mfrow=c(1,3))

hist(runFWtestResultsProdSHighp1510[1,]-1, main = "Error Distribution for Task with Weight=1",
xlab="Error for Weight 1",xlim=c(0,.3),xaxt='n')

axis(side=1, at=seq(0,0.3,by=.05), labels=paste(seq(0,30,by=5),'%'))

hist((runFWtestResultsProdSHighp1510[2,]-5)/5, main = "Error Distribution for Task with Weight=5",
xlab="Error for Weight 5",xlim=c(-.3,.3),xaxt='n')

axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))

hist((runFWtestResultsProdSHighp1510[3,]-10)/10, main = "Error Distribution for Task with Weight=10",
xlab="Error for Weight 10",xlim=c(-.3,.3),xaxt='n')

axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))



runFWtestProdSHighp123<-function(n) { #this function runs findWeightsprodSame() n times with
weights 1, 2, and 3.  Uses 50 performers

ansMat<-matrix(nrow=3,ncol=n)

for(i in 1:n){

thisAns<-findWeightsProdSameHighp(50,1,2,3)

ansMat[1,i]<-thisAns[1]
```

```
ansMat[2,i]<-thisAns[2]
ansMat[3,i]<-thisAns[3]
}
return(ansMat)
}


runFWtestResultsProdSHighp123<-runFWtestProdSHighp123(100)

mean(runFWtestResultsProdSHighp123[1,])
mean(runFWtestResultsProdSHighp123[2,])
mean(runFWtestResultsProdSHighp123[3,])

quantile(runFWtestResultsProdSHighp123[1,],.05)
quantile(runFWtestResultsProdSHighp123[2,],.05)
quantile(runFWtestResultsProdSHighp123[3,],.05)

quantile(runFWtestResultsProdSHighp123[1,],.95)
quantile(runFWtestResultsProdSHighp123[2,],.95)
quantile(runFWtestResultsProdSHighp123[3,],.95)

mean(abs(runFWtestResultsProdSHighp123[1,]-1))/1
mean(abs(runFWtestResultsProdSHighp123[2,]-2))/2
mean(abs(runFWtestResultsProdSHighp123[3,]-3))/3


par(mfrow=c(1,3))
hist(runFWtestResultsProdSHighp123[1,]-1, main = "Error Distribution for Task with Weight=1",
xlab="Error for Weight 1",xlim=c(0,.3),xaxt='n')
```

```
axis(side=1, at=seq(0,0.3,by=.05), labels=paste(seq(0,30,by=5),'%'))

hist((runFWtestResultsProdSHighp123[2,]-2)/2, main = "Error Distribution for Task with Weight=2",
xlab="Error for Weight 2",xlim=c(-.3,.3),xaxt='n')

axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))

hist((runFWtestResultsProdSHighp123[3,]-3)/3, main = "Error Distribution for Task with Weight=3",
xlab="Error for Weight 3",xlim=c(-.3,.3),xaxt='n')

axis(side=1, at=seq(-0.3,0.3,by=.1), labels=paste(seq(-30,30,by=10),'%'))
```

Code for Analysis with Varying Productivity

```
perfDataGenProdVary<-function(n,prodMat,w1,w2,w3){ #This function will produce production data for
n peformers with work items with actual weights w1, w2, and w3.  Productivity for each performer is
stored in a vector of length n called prodMat

prime=0

ansMat<-matrix(nrow=3,ncol=n) #create a matrix to store a column of information for each performer
with the number of each of the three work items produced

for(i in 1:n) { #run a loop for each performer to assign credit and calculate scores and proportions

capacity = prodMat[i] #set capacity according to productivity form prodMat

score = 0 + prime #set score

WI1<-0 #initiate work item 1 count to 0

WI2<-0 #initiate work item 2 count to 0

WI3<-0 #initiate work item 3 count to 0

while(capacity>0){ #run a while loop while there is still capacity left

thisWI<-floor(runif(1,1,4)) #generate a random integer between 1 and 3

if (thisWI==1) { #evaluate which item was randomly produced and increment that WI count

capUse <- w1

WI1<-WI1+1

} else if (thisWI==2) {
```

```
capUse <- w2
WI2<-WI2+1
} else {
capUse <- w3
WI3<-WI3+1
}
capacity = capacity - capUse #remove the capacity associated with this work item

if(capacity<0){ #if you have run over capacity, reverse the WI and score increment
if (thisWI==1) {
WI1<-WI1-1
} else if (thisWI==2) {
WI2<-WI2-1
} else {
WI3<-WI3-1
}
}

}
ansMat[,i]=c(WI1,WI2,WI3) #update the ith row with the three score counts
prime<-prime+.0001
} #close the for loop
return (ansMat)
} #close the function


findWeightsProdVary<-function(n,w1,w2,w3){ #create a function that finds the weights given n performers and actual weights of w1, w2, and w3.  The goal is to get w1, w2, and w3
wght1<-1 #initialize the weights to 1
```

```
wght2<-1

wght3<-1

counter <-0 #create a counter so the while loop doesn't run forever

prodMatrix<-rnorm(n,mean=100,sd=15) #create a vector of productivities length n using normal distribution

results<-perfDataGenProdVary(n,prodMatrix,w1,w2,w3)

while(TRUE){ #run a while loop to refine the weightings

scores<-wtdScoreCalc(n,results,wght1,wght2,wght3) #generate the weighted scores

cor1<-cor(scores,results[1,]) # calculate the correlations and p values

test1<-cor.test(scores,results[1,])

p1<-test1$p.value


cor2<-cor(scores,results[2,])

test2<-cor.test(scores,results[2,])

p2<-test2$p.value


cor3<-cor(scores,results[3,])

test3<-cor.test(scores,results[3,])

p3<-test3$p.value


corSum<-0 #create a variable for the sum of all the p's

corCount<-0 #create a variable for the count of all the p's


p1bool<-FALSE #create boolean values to indicates whether or not the p-values are less than .05

p2bool<-FALSE

p3bool<-FALSE


if(p1<.5){ #if the p-value of the correlation coefficient for wi1 is less than .5 then add it to the sum and trigger the pvalue boolean variable
```

```
corSum<-corSum+cor1

corCount<-corCount+1

p1bool<-TRUE

}


if(p2<.5){ #if the p-value of the correlation coefficient for wi2 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor2

corCount<-corCount+1

p2bool<-TRUE

}


if(p3<.5){ #if the p-value of the correlation coefficient for wi3 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor3

corCount<-corCount+1

p3bool<-TRUE

}


corAvg<-corSum/corCount #calculate the average correlation coefficient using the coefficients that had p-values less than .5


if(p1bool){ #if the p-value for cor1 was less than .5, change weighting1

if(cor1<corAvg){

wght1<-wght1*1.01

}

else {wght1<-wght1*.99}

}
```

```
if(p2bool){ #if the p-value for cor2 was less than .5, change weighting2
if(cor2<corAvg){
wght2<-wght2*1.01
}
else {wght2<-wght2*.99}
}


if(p3bool){ #if the p-value for cor3 was less than .5, change weighting3
if(cor3<corAvg){
wght3<-wght3*1.01
}
else {wght3<-wght3*.99}
}

counter<-counter+1

if( ((!p1bool & !p2bool) & !p3bool) | (counter==1000) ){
break
}

}
minW<-min(wght1,wght2,wght3) #get the minimum weight to index the weights to 1
return(c(wght1/minW,wght2/minW,wght3/minW))
}


runFWtestProdV123<-function(n) { #this function runs findWeightsprodvary() n times with weights 1, 2, and 3 with 50 performers
```

```
ansMat<-matrix(nrow=3,ncol=n)
for(i in 1:n){
thisAns<-findWeightsProdVary(50,1,2,3)
ansMat[1,i]<-thisAns[1]
ansMat[2,i]<-thisAns[2]
ansMat[3,i]<-thisAns[3]
}
return(ansMat)
}

runFWtestResultsProdV123<-runFWtestProdV123(100)

mean(runFWtestResultsProdV123[1,])
mean(runFWtestResultsProdV123[2,])
mean(runFWtestResultsProdV123[3,])

quantile(runFWtestResultsProdV123[1,],.05)
quantile(runFWtestResultsProdV123[2,],.05)
quantile(runFWtestResultsProdV123[3,],.05)

quantile(runFWtestResultsProdV123[1,],.95)
quantile(runFWtestResultsProdV123[2,],.95)
quantile(runFWtestResultsProdV123[3,],.95)

mean(abs(runFWtestResultsProdV123[1,]-1))/1
mean(abs(runFWtestResultsProdV123[2,]-2))/2
mean(abs(runFWtestResultsProdV123[3,]-3))/3
```

```
par(mfrow=c(1,3))

hist(runFWtestResultsProdV123[1,]-1, main = "Error Distribution for Task with Weight=1", xlab="Error for Weight 1",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))

hist((runFWtestResultsProdV123[2,]-2)/2, main = "Error Distribution for Task with Weight=2", xlab="Error for Weight 2",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))

hist((runFWtestResultsProdV123[3,]-3)/3, main = "Error Distribution for Task with Weight=3", xlab="Error for Weight 3",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))
```

Code for Revised Weighting Capacity Analysis to Account for Varying Productivity

```
findWeightsProdVaryRev<-function(n,w1,w2,w3){ #create a function that finds the weights given n performers and actual weights of w1, w2, and w3.  The goal is to get w1, w2, and w3

wght1<-1 #initialize the weights to 1

wght2<-1

wght3<-1

counter <-0 #create a counter so the while loop doesn't run forever

prodMatrix<-rnorm(n,mean=100,sd=15) #create a vector of productivities length n using normal distribution

results<-perfDataGenProdVary(n,prodMatrix,w1,w2,w3)


prodScores<-rep(1,n) #initialize all productivity scores to 1


while(TRUE){ #run a while loop to refine the weightings

scores<-wtdScoreCalc(n,results,wght1,wght2,wght3)/prodScores #generate the weighted scores and divide by their productivity scores (all 1, initially)

cor1<-cor(scores,results[1,]) # calculate the correlations and p values

test1<-cor.test(scores,results[1,])

p1<-test1$p.value
```

```
cor2<-cor(scores,results[2,])
test2<-cor.test(scores,results[2,])
p2<-test2$p.value

cor3<-cor(scores,results[3,])
test3<-cor.test(scores,results[3,])
p3<-test3$p.value

corSum<-0 #create a variable for the sum of all the p's
corCount<-0 #create a variable for the count of all the p's

p1bool<-FALSE #create boolean values to indicates whether or not the p-values are less than .05
p2bool<-FALSE
p3bool<-FALSE

if(p1<.5){ #if the p-value of the correlation coefficient for wi1 is less than .5 then add it to the sum and trigger the pvalue boolean variable
corSum<-corSum+cor1
corCount<-corCount+1
p1bool<-TRUE
}

if(p2<.5){ #if the p-value of the correlation coefficient for wi2 is less than .5 then add it to the sum and trigger the pvalue boolean variable
corSum<-corSum+cor2
corCount<-corCount+1
p2bool<-TRUE
}
```

```
if(p3<.5){ #if the p-value of the correlation coefficient for wi3 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor3

corCount<-corCount+1

p3bool<-TRUE

}


corAvg<-corSum/corCount #calculate the average correlation coefficient using the coefficients that had p-values less than .5


if(p1bool){ #if the p-value for cor1 was less than .5, change weighting1

if(cor1<corAvg){

wght1<-wght1*1.01

}

else {wght1<-wght1*.99}

}



if(p2bool){ #if the p-value for cor2 was less than .5, change weighting2

if(cor2<corAvg){

wght2<-wght2*1.01

}

else {wght2<-wght2*.99}

}



if(p3bool){ #if the p-value for cor3 was less than .5, change weighting3

if(cor3<corAvg){
```

```
wght3<-wght3*1.01
}
else {wght3<-wght3*.99}
}

counter<-counter+1

if( ((!p1bool & !p2bool) & !p3bool) | (counter==10000) ){
break
} #break the while loop

} #rerun the while loop

prodScores<-wtdScoreCalc(n,results,wght1,wght2,wght3) #update the productivity scores with the new weights
#Now run the whole thing again with a second month of data using the prodScores!
results<-perfDataGenProdVary(n,prodMatrix,w1,w2,w3)
counter=0 #reset the counter

while(TRUE){ #run a while loop to refine the weightings
scores<-wtdScoreCalc(n,results,wght1,wght2,wght3)/prodScores #generate the weighted scores and divide by their productivity scores
cor1<-cor(scores,results[1,]) # calculate the correlations and p values
test1<-cor.test(scores,results[1,])
p1<-test1$p.value

cor2<-cor(scores,results[2,])
test2<-cor.test(scores,results[2,])
p2<-test2$p.value
```

```
cor3<-cor(scores,results[3,])

test3<-cor.test(scores,results[3,])

p3<-test3$p.value


corSum<-0 #create a variable for the sum of all the p's

corCount<-0 #create a variable for the count of all the p's


p1bool<-FALSE #create boolean values to indicates whether or not the p-values are less than .05

p2bool<-FALSE

p3bool<-FALSE


if(p1<.5){ #if the p-value of the correlation coefficient for wi1 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor1

corCount<-corCount+1

p1bool<-TRUE

}


if(p2<.5){ #if the p-value of the correlation coefficient for wi2 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor2

corCount<-corCount+1

p2bool<-TRUE

}


if(p3<.5){ #if the p-value of the correlation coefficient for wi3 is less than .5 then add it to the sum and trigger the pvalue boolean variable

corSum<-corSum+cor3

corCount<-corCount+1
```

```
p3bool<-TRUE
}

corAvg<-corSum/corCount #calculate the average correlation coefficient using the coefficients that had p-values less than .5

if(p1bool){ #if the p-value for cor1 was less than .5, change weighting1
if(cor1<corAvg){
wght1<-wght1*1.01
}
else {wght1<-wght1*.99}
}

if(p2bool){ #if the p-value for cor2 was less than .5, change weighting2
if(cor2<corAvg){
wght2<-wght2*1.01
}
else {wght2<-wght2*.99}
}

if(p3bool){ #if the p-value for cor3 was less than .5, change weighting3
if(cor3<corAvg){
wght3<-wght3*1.01
}
else {wght3<-wght3*.99}
}
```

```
counter<-counter+1

if( ((!p1bool & !p2bool) & !p3bool) | (counter==1000) ){
break
} #break the while loop

} #rerun the while loop
minW<-min(wght1,wght2,wght3) #get the minimum weight to index the weights to 1
return(c(wght1/minW,wght2/minW,wght3/minW))
}

runFWtestProdV123<-function(n) { #this function runs findWeightsprodvary() n times with weights 1, 2, and 3
ansMat<-matrix(nrow=3,ncol=n)
for(i in 1:n){
thisAns<-findWeightsProdVaryRev(50,1,2,3)
ansMat[1,i]<-thisAns[1]
ansMat[2,i]<-thisAns[2]
ansMat[3,i]<-thisAns[3]
}
return(ansMat)
}

runFWtestResultsProdV123<-runFWtestProdV123(100)

mean(runFWtestResultsProdV123[1,])
mean(runFWtestResultsProdV123[2,])
mean(runFWtestResultsProdV123[3,])
```

```
quantile(runFWtestResultsProdV123[1,],.05)

quantile(runFWtestResultsProdV123[2,],.05)

quantile(runFWtestResultsProdV123[3,],.05)


quantile(runFWtestResultsProdV123[1,],.95)

quantile(runFWtestResultsProdV123[2,],.95)

quantile(runFWtestResultsProdV123[3,],.95)


mean(abs(runFWtestResultsProdV123[1,]-1))/1

mean(abs(runFWtestResultsProdV123[2,]-2))/2

mean(abs(runFWtestResultsProdV123[3,]-3))/3


par(mfrow=c(1,3))

hist(runFWtestResultsProdV123[1,]-1, main = "Error Distribution for Task with Weight=1", xlab="Error for Weight 1",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))

hist(runFWtestResultsProdV123[2,]-2, main = "Error Distribution for Task with Weight=2", xlab="Error for Weight 2",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))

hist(runFWtestResultsProdV123[3,]-3, main = "Error Distribution for Task with Weight=3", xlab="Error for Weight 3",xlim=c(-2,2),xaxt='n')

axis(side=1, at=seq(-2,2,by=.5), labels=paste(seq(-200,200,by=50),'%'))
```