Evaluating New Matrix Pooled Testing Methods for Detecting HIV Treatment Failure with and without Covariate Information

Adam Brand

A thesis
submitted in partial fulfillment
requirements for the degree of

Master of Science

University of Washington
2016

Committee:
Susanne May
James Hughes

Program Authorized to Offer Degree
Biostatistics

University of Washington

Abstract

Evaluating New Matrix Pooled Testing Methods for Detecting HIV Treatment Failure with and without Covariate Information

Adam Brand

Chair of the Supervisory Committee:
Susanne May, Associate Professor
Department of Biostatistics

Antiretroviral treatment for HIV has proven to lengthen and improve patients' lives, and greatly reduce the risk of transmission. Patients receiving antiretroviral treatment for HIV must be monitored for treatment failure, so the treatment regimen can be altered to maintain low HIV viral load levels. Monitoring/testing treated HIV patients requires resources not available to resource-limited regions most in need. It has been shown that matrix pooled testing can be efficient compared to individual testing under certain conditions however, further improvements are needed. The current best-performing matrix pooled testing method for detecting HIV treatment failure does not use covariate information. Other methods which do use covariate information have not yet been shown to outperform this method in settings with realistic, skewed viral load values. We propose new methods of matrix pooled testing, some of which use covariate information, and evaluate method performance with respect to relative efficiency, sensitivity and number of testing rounds. Based on simulation results we identify a method which incorporates covariate information and outperforms the current best-performing method in settings using realistic, skewed viral load values when we have access to a predictor(s) of virologic failure.

Table of Contents

1. Introduction

In 2014 approximately 36.9 million people were HIV infected, there were 2 million new cases of HIV and 1.2 million HIV-related deaths worldwide. About 70% of HIV infected people live in sub-Saharan Africa. Almost 60% of people infected with HIV are not receiving antiretroviral therapy (ART) likely because their communities do not have the resources required to either diagnose them or dispense treatment and monitor patients for treatment failure. Proper treatment for HIV can reduce transmission by up to 96% according to the landmark RCT, HPTN 052 (Cohen et al., 2011). By allowing 60% of the HIV infected population to go untreated we are ignoring an opportunity to stop the spread of HIV. Cheaper, effective ways of treating and monitoring treatment for all HIV infected individuals are essential to eradicating HIV.

As HIV replicates within the patient it can mutate and become resistant to particular medications. When a patient develops clinical resistance to their particular ART regimen they will need to have their ART medications switched for non-resistant ones. Virologic testing can be crucial to ensuring that a patient's viral load (VL; amount of HIV in blood) remains controlled. Therefore all HIV treated patients must undergo regular, continual virologic testing. The difficulty in providing virologic testing to all HIV patients is that virologic testing is expensive and not available everywhere (roughly $250 per test). Of the utmost importance in controlling the spread of HIV is that cheaper, effective methods of virologic testing are developed which can be used in resource poor regions.

One approach for monitoring HIV viral load that can be used in resource poor settings is pooled testing. Pooled testing occurs when multiple blood samples are combined and tested simultaneously in order to gain knowledge about each individual in a group of patients. The aim is that by testing groups of patients, fewer tests will be required than testing each patient

separately. Blood banks have used pooled testing successfully to screen out blood infected with transmittable diseases (Bilder, Tebbs and Chen, 2010). The efficiency gained by such testing depends on the prevalence of the disease.

Pooled testing methods have been evaluated for identifying acute HIV infection among other non-infected samples [(Behets et al., 1990; Brookmeyer R., 1999; Busch M.P. (2005); Cahoon-Young et al., 1989; Gastwirth and Hammick, 1989; Hammick and Gastwirth, 1994; Kim et al., 2007; Kline et al., 1989; Patterson et al., 2007; Pilcher et al., 2005; Pilcher et al., 2002; Quinn et al., 2000; Tu, Litvak and Pagano, 1995; Westreich et al., 2008).] Many of these methods illustrate the utility of pooled testing for acute HIV infection. However, these methods are not useful for identifying treatment failure among a population where every individual is HIV positive, because they are not always able to distinguish between HIV infected patients experiencing treatment failure and those who are not. The higher prevalence of treatment failure and possible similarity in HIV viral load for patients experiencing treatment failure and those not necessitate measuring the amount of HIV viral load (VL) in a patient's blood. Pooled testing methods which account for a continuous measure of HIV viral load have also been evaluated (May et al., 2010; Hanscom, 2014; Hanscom, May and Hughes, 2014).

When using HIV viral load as a measure of treatment failure/success a failure threshold, *t*, must be defined. Each individual whose viral load is above *t* is considered to be experiencing treatment failure. The goal of pooled testing for detecting HIV treatment failure is to identify every individual experiencing treatment failure with as few tests and testing rounds as possible. Therefore, method performance is compared to testing each individual separately with respect to relative efficiency, sensitivity and number of testing rounds. Relative efficiency is the percentage of tests saved over individual testing. Sensitivity is the percentage of treatment

failures identified compared to individual testing. Number of testing rounds indicates how many batches of tests are required to classify an entire matrix of individuals, and represents testing turnaround-time/cost.

May et al. (2010) evaluated 3 different pooling methods and showed that the use of quantitative viral load (VL) information can lead to efficient pooling approaches in this setting using realistically skewed viral load values. One of these methods is the mini + algorithm method (Mini). The Mini method pools *n* samples, and tests the pool to get an average viral load of every contributing sample. If the pool tests less than *t/n*, every individual contributing to the pool is classified as not experiencing treatment failure. If the pool tests greater than *t/n*, individuals are tested one-at-a-time until the updated pool average (after subtracting the tested sample's viral load divided by *n*) is less than *t/n*.

Matrix pooled testing methods arrange individual samples into an *n x n* matrix, and then pool samples across rows/columns which, when tested, provide average viral load values for each row/column. If a row/column average tests below *t/n* all individuals contributing to that row/column average are considered non-treatment failures. Based on the row/column averages (and covariate information if applicable), matrix pooled testing methods choose which individuals to test and how many tests to conduct for each testing round. As individuals are tested, their observed viral load value divided by *n* is subtracted from the corresponding row and column averages. Testing continues until each individual is classified as either experiencing treatment failure or not. Performance of pooled testing methods depend on how many and which individuals are chosen for testing each round.

May et al. (2010) further showed that a matrix pooled testing method, the simple search (SS), using a *10 x 10* matrix design can improve on the efficiency of non-matrix pooled testing

algorithms and matrix designs of other sizes for treatment failure prevalences from 1%-25%. The simple search is a matrix pooled testing method which chooses to test the one individual sample corresponding to the highest row and column average per testing round.

Hanscom (2014; Hanscom, May and Hughes, 2014) extended these methods further, and evaluated additional matrix pooled testing methods; the Modified Simple Search (MSS), an EM-based algorithm (EM), and the Model Based Search (MBS). The EM-based algorithm is an iterative matrix pooled testing method which alternates between estimating treatment failure prevalence using Expectation-Maximization (EM), and choosing to test individual samples most likely to be above the failure threshold. Initial distributional parameters such as failure prevalence are estimated using pre-existing data and scientific knowledge. The MBS method is another iterative matrix pooled testing method, alternating between estimating the failure status of each individual using Monte Carlo Markov Chains (MCMC) and testing the samples most likely to be above the failure threshold. Covariate information can be incorporated in the MBS method in order to more accurately estimate individual failure status. While the EM and MBS methods performed well when HIV viral load values were normally distributed and covariate information was strongly predictive, results were less consistent when applied to real data with highly-skewed viral load values.

The modified simple search (MSS) is a matrix pooled testing method similar to the SS, and is currently the best-performing, most-reliable pooled testing method for detecting HIV treatment failure in real settings. The MSS chooses to test the $n/2$ samples per round corresponding to the highest row and column averages with the added restriction that each sample tested correspond to distinct rows and columns (this restriction is relaxed when there are few individuals unclassified). The MSS method outperforms the SS method with respect to

efficiency and number of testing rounds, and performs similarly to the SS with respect to sensitivity. The EM-based algorithm outperforms the MSS method in all 3 performance categories with normally distributed viral load values. However, when using realistically-skewed viral load values, the sensitivity of the EM-based algorithm is much lower than the MSS method's sensitivity (10% or more difference). The MSS method also performs more reliably with respect to sensitivity than the MBS method when covariate information was missing or not strongly predictive of treatment failure in real settings. The Mini method, while not as efficient as the MSS method, maintains high sensitivity at all failure prevalences. These methods illustrate the potential for meaningful efficiency gains using matrix pooled testing methods. Currently no method has been shown to outperform the MSS method in a realistic setting using skewed viral load values.

In this paper we propose new matrix pooled testing methods for detecting HIV treatment failure with and without the use of covariate information, and evaluate the performance of these methods by comparing them to the current best-performing method for detecting HIV treatment failure, the MSS method. The next section includes a description of 3 new pooled testing methods for detecting HIV treatment failure and 2 established methods for comparison. Results and a discussion are presented next followed by an appendix which includes all R code for reproducibility.

2. Methods

2.1 *Matrix Pooled Testing Design Setup*

All matrix pooled testing methods compared in this paper utilize a 10 x 10 matrix design with each entry corresponding to an individual sample. We chose this design for method comparison; because previous research (May et al., 2010; Hanscom, 2014; Hanscom, May and Hughes, 2014)

suggest that this design is optimal for viral load values and treatment failure prevalences seen in practice. In simulations we assume that the individual samples contribute equally to the row/column pools; and thus represent an actual average of the viral load values in the corresponding rows/columns.

For all methods we define a constant viral load failure threshold of t = 1000 copies/mL. Individuals with a viral load greater than 1000 copies/mL are classified as treatment failures. Since the row and column pools are estimates of the average of all samples in that row/column, we define a failure threshold for rows and columns as t/n = 100. If a row or column tests below 100 copies/mL, we classify all individuals contributing to that pool as not experiencing treatment failure. This is assessed at the beginning of testing and between testing rounds as testing iterations progress.

Method performance is evaluated using 3 criteria; relative efficiency, sensitivity and number of testing rounds. Because our matrix design tests groups of 100 patients, 1 for each cell in the 10 x 10 testing matrix, a relative efficiency of 42% means that we were able to classify all patients using 58 tests. Due to measurement error the true sensitivity of any method can never be known in practice, so sensitivity is defined relative to the individual test result. Our performance estimates are based on the mean efficiency, sensitivity and number of testing rounds over 500 simulated matrices corresponding to 50,000 patients tested.

2.2 *Current Best-Performing Methods*

The 2 benchmarks for known, best-performing methods are the modified simple search (MSS) and the minipool + algorithm (Mini) initially proposed by May et al. (2010). The MSS method is the best-performing matrix pooled testing method for detecting HIV treatment failure in terms of efficiency and number of testing rounds while maintaining high sensitivity. The MSS method

does not use covariate information, and it has been shown to outperform the MBS method, which uses covariate information, in practical settings (Hanscom, 2014; Hanscom, May and Hughes, 2014). Implemented using a 10 x 10 matrix design; the MSS method tests the 5 samples corresponding to the highest row and column averages each testing round. The 5 samples are chosen so they correspond to distinct rows/columns. Restricting the individuals tested to those in distinct rows and columns provides more information about the matrix than without this restriction; allowing us to learn about the entire matrix and incorporate this knowledge with each round of testing. When there are few individuals remaining to be classified as either treatment failures or non-treatment failures the restrictions on the number of samples tested and how they are chosen are relaxed. A more detailed description of all methods evaluated in this paper are included in the appendix.

The Mini method is implemented by combining 10 minipools of size 10 (100 patients) in order to fairly compare the Mini method to matrix pooling methods. Each of the 10 minipools is tested to obtain viral load averages. If a pool tests less than 100 copies/mL, every individual contributing to that pool is considered a non-failure. Each testing round one individual is chosen at random and tested from each of the minipools whose viral load average is greater than 100, and those viral loads are subtracted from the corresponding minipool totals until all patients are classified. The Mini method is therefore like a matrix design method which only pools and tests across rows. This gives the Mini method some interesting features. The maximum number of rounds for any 100 patients is 11; one for the initial pool tests and a maximum of 10 individual testing rounds. Also, the individuals tested within each pool are chosen at random instead of being chosen due to the combination of row and column averages. As you will see this leads to decreased efficiency, but high sensitivity at all prevalences. The Mini method performance will

allow us to evaluate the 'cost' of our efficiency gains, i.e., the amount of sensitivity we give up for greater efficiency.

The MSS and Mini methods provide a benchmark for improvement. Each of these methods seem to perform well in settings with skewed viral load values which is characteristic of viral load values. However, it seems reasonable to assume that incorporating more information, such as covariate information, can produce new methods which outperform these existing methods.

2.3 *Methods that use Covariate information*

We introduce 3 new methods that incorporate covariate information: the minipool with covariates method (Mini/Cov), the linear regression method (Linreg) and the linear regression systems of equations method (LRSOE). These methods use covariate information to predict each sample's viral load, and combine those predictions with the row and column averages to determine how many and which individuals to test in the next round. The following study was used to inform the associations between covariate information and viral load for simulations. Robbins et al. (2007) found that poor ART adherence and having experienced a prior treatment failure were the strongest predictors of HIV treatment failure in patients whose HIV viral load was previously suppressed. The hazard ratios for experiencing virologic failure after being suppressed were estimated to be 3.62 (95% CI: 2.27-5.79) and 2.08 (95% CI: 1.22-2.39) for poor adherence and prior failure, respectively, while adjusting for each other and other possible confounding factors. This suggests poor adherence and prior failure status as accessible covariate information that is predictive of HIV treatment failure.

The difficulty in creating a predictive model for HIV viral load is that hazard ratios cannot be directly converted into an effect estimate on the continuous measurement of HIV viral

load. The purpose of this work is not to identify a true HIV viral load prediction model, but to answer whether or not we can improve on the MSS method performance in the setting where predictors of HIV viral load are known and accessible. Therefore, we assume that we have access to known predictors of HIV viral load, and propose the following prediction model using poor adherence (Adh) and prior failure status (PFS):

$$\log_{10}(VL) = 1 + .05(100 - Adh) + PFS + .05(100 - Adh)*PFS \quad (1)$$

Adh is self-reported adherence on a 0 to 100 scale with 100 being perfect adherence. PFS stands for prior failure status, and represents whether or not the patient has experienced a prior treatment failure. PFS is binary, and should be accessible through the patient's medical records.

We have chosen the coefficients for our prediction model to reflect a gradual rise in HIV viral load for poorer adherence and prior failure status. There is no empirical evidence for the chosen coefficients however, scientific knowledge suggests that these coefficients describe a reasonable prediction model. Noticing that model (1) is on the log10 scale for viral load this model predicts a viral load of 10 for a perfectly adherent patient who has not experienced a prior treatment failure and a viral load of 100 for a perfectly adherent patient who has experienced a prior treatment failure. Therefore, we only predict a patient to be experiencing treatment failure if their adherence is less than 60 for patient without prior treatment failure and less than 90 for a patient with prior treatment failure. We include an interaction which enhances the effect of poor adherence on viral load level for patients with prior treatment failure. If a patient has experienced a prior treatment failure then it night be reasonable to assume that their particular HIV strain has undergone more mutations that a patient without prior treatment failure, and therefore we might expect the PFS patient to become resistant to new medication easier. The

proposed methods which use versions of model (1) for performance evaluation are described below.

The Mini/Cov method is a basic extension of the Mini method. The testing is conducted similarly by forming 10 minipools of 10 samples each. The extension is that we use linear regression on known covariates to estimate each individual's viral load. Instead of randomly choosing to test an individual in an unclassified row, we choose to test the individual in that row with the highest predicted viral load based on our regression model. If there are ties for the highest predicted viral load, which occurs in practice, an individual is selected randomly from among those tied.

The linear regression method (Linreg) is a matrix pooled testing method, and is constructed as described above. In choosing which samples to test it first predicts each individual's viral load. There is no guarantee that the predicted values will agree with the row and column averages, so we proportionally alter our predictions in an iterative fashion to fit our predictions to the observed row and column averages (further detail on this and all evaluated methods in the appendix). The Linreg method finds the solution to the testing matrix which most closely agrees with our viral load predictions and the observed row and column averages. It then tests 5 samples corresponding to the 5 highest viral load values from our solved matrix. After each round of testing the information from the tests is incorporated, and the entire process repeats until all patients are classified.

The linear regression systems of equations (LRSOE) method combines aspects of the Linreg method and a systems of equations method (SOEM; discussed briefly later). Like the Linreg method, LRSOE predicts each individual's viral load based on known covariate information. Then, starting with the individual with the highest predicted viral load, it assigns

the highest possible viral load value based on the row and column averages for that individual. For example if the row average is 500 and the column average is 600 for that sample, we can choose to assign a maximum of 500x10, or 5000, to that individual matrix cell. Notice that we cannot assign any value greater than 5000 to this cell, because that would necessarily make the corresponding row average greater than the observed 500. (Note: this assumes that every other individual in the same row has a viral load of zero. In simulations we define 50 copies/mL as the lower limit of detection, so if no other patient in the row has been classified, we assign 5000 – 450, or 4550, to the selected matrix cell and 50 to every other cell in that row.) We proceed in this way to obtain a possible solution to the matrix. Then we test every sample with a viral load higher than the lower limit of detection according to our matrix solution. Because LRSOE chooses how many individuals to test, the number of tests varies from round to round and depends on that round's solution to the matrix. As with other matrix pooled testing methods, all individuals contributing to a row/column whose average viral load is less than 100 copies/mL are classified as non-treatment failures. After each round of testing the entire process repeats with updated viral load information until all individuals are classified as either treatment failures or non-treatment failures.

These three new methods were evaluated and compared to the current best-performing methods. We also evaluated other methods (results not shown due to clearly inferior performance); some which use covariate information and some which do not. The next section will briefly discuss these methods.

2.4 *Other Explored Methods*

This section briefly discusses other methods evaluated (with inferior performance), so they can be ruled out as possible improvements. The systems of equations minimization (SOEM) method

is one such method, and does not use covariate information. SOEM finds the solution to the testing matrix which minimizes the number of individuals experiencing treatment failure, and tests those samples. The new knowledge is incorporated and the process repeats until all individuals are classified. Another variation of SOEM is the systems of equations (SOE) method. The SOE method is identical to SOEM except that the solution to the testing matrix does not necessarily minimize the number of treatment failures in a given matrix. Instead, SOE assigns viral load values proportionally based on every row and column average. We evaluated both SOE and SOEM with and without a test-per-round limit.

The LRSOE Limit method imposes a test limit of 5 per round on our previously mentioned LRSOE method.

Finally, we evaluated 2 other combinations of the systems of equations and linear regression methods: systems of equations minimization linear regression (SOEMLR) and systems of equations linear regression (SOELR). These methods solve the systems of equations and use covariate information to predict each individual's viral load in separate steps. Then viral load predictions are converted into weights which are then multiplied by the systems of equations solutions and the highest resulting values are tested first. We felt it important to describe these methods which do not show promise, so that they can be excluded from further study.

2.5 *Simulation Studies*

2.5.1 *Data Generation*

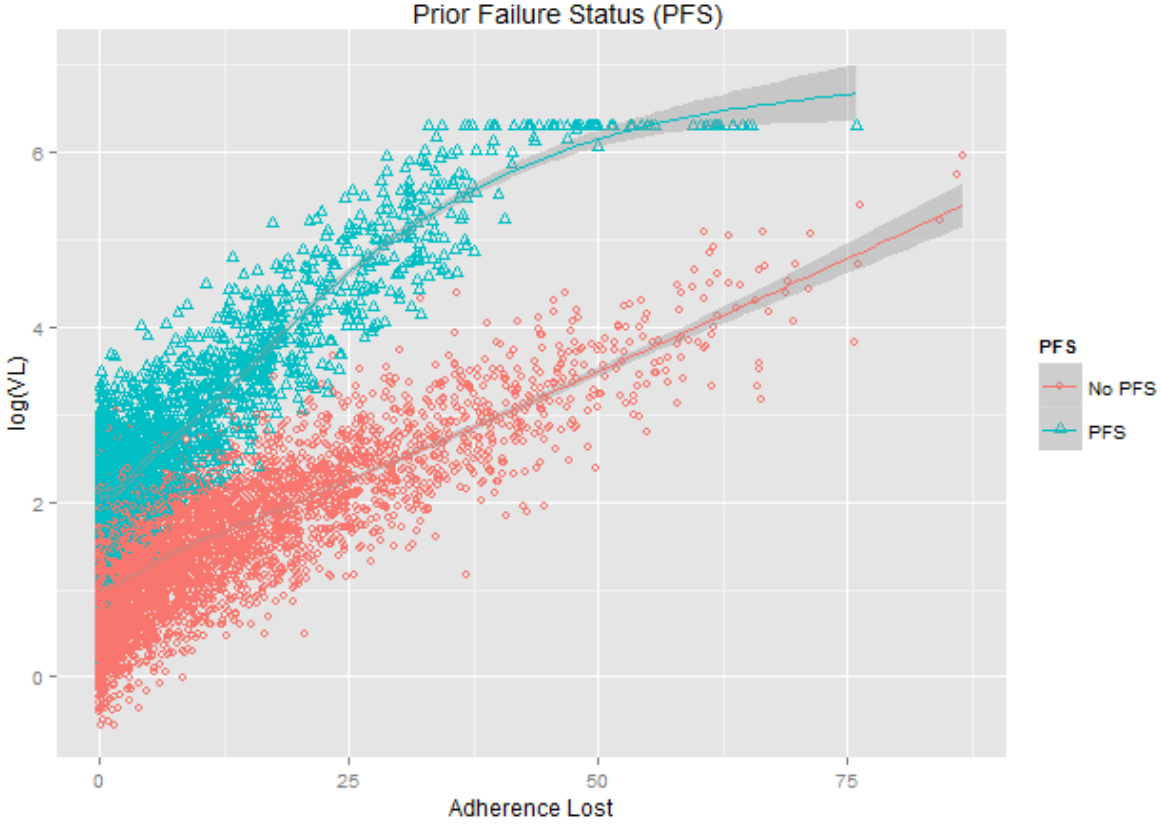
One set of 1 million patient data was simulated for each of 4 approximate prevalences of treatment failure; 1%, 5%, 10% and 20%. These data were simulated using the following model where VL, Adh and PFS are defined as in model (1):

$\log_{10}(VL) = 1 + .05(100 - Adh) + PFS + .05(100 - Adh)*PFS + \varepsilon$ where $\varepsilon \sim N(0, 0.5)$.

An important consideration for performance evaluation is producing skewed viral load values to obtain an accurate estimate of how these methods perform with likely viral load values. May et al. (2010) describe a mixed distribution which mimics actual HIV viral load values seen in practice. For HIV viral load values under 500 copies/mL 85%, 5% and 10% of values follow uniform distributions (0,50), (50,100) and (100,500) respectively. For values over 500 copies/mL, 93% of viral load values follow a gamma distribution with shape parameter of 1.6 and a scale parameter of 0.5 on the log10 scale, shifted 2.7 in the positive direction (or equivalently, 500 on the actual viral load scale). The other 7% of values over 500 copies/mL follow a gamma distribution also on the log10 scale, shifted 2.7 and scale parameter 0.5, but with a shape parameter of 3.18.

When simulating data we generate covariate values first, and then use our model to generate the outcome. The difficulty is ensuring that the outcome, log10 viral load, has the mixed distribution likely seen in practice based on the distributions we choose for our covariates. Therefore, we wrote a shiny application (included in appendix) to obtain a visual comparison of the mixed viral load distribution proposed by May et al. (2010) and the viral load distribution created from our data generating model. The shiny application allows us to change parameters of our covariate distributions in order to match the viral load outcomes from our model to those seen in practice; paying particular attention to match the viral load values surrounding and over the cutoff of 1000 copies/mL. In this way we build a model predictive of HIV viral load which produces realistic, skewed viral load values. Figure 1 presents an example of 10,000 true viral load values sampled randomly from the generated dataset of 1 million patients with treatment failure prevalence of 10% and prior failure prevalence of 25%.

FIGURE 1: 10,000 simulated viral load values on the log10 scale with 10% Treatment Failure Prevalence and 25% Prior Failures by Prior Failure Status (PFS)



In Figure 1, viral load values are truncated at 2,000,000 copies/mL, the upper limit of detection for current testing equipment.

All four datasets (with treatment failure prevalences of approximately 1%, 5%, 10% and 20%) were generated to have 25% prior failure status, generated as independent Bernoulli distributions with probability 0.25. The adherence data was also generated independently as beta distributions with parameters (8, 0.05), (7, 0.15), (4, 0.33) and (6, 1.05) for treatment failure prevalences of 1%, 5%, 10% and 20% respectively, and multiplied by 100. These parameters produce treatment failure prevalences which are within 0.5% of our desired failure prevalences. It is probably reasonable to assume that adherence and prior treatment failure are not

independent. In reality however, this should have little effect, if any, on our estimates of method performance in the setting where we have covariate information predictive of HIV viral load.

2.5.2 *Method Evaluation*

We simulated and classified 500 matrices, representing 50,000 patients, for all methods under eight different scenarios for each prevalence of 1%, 5%, 10% and 20%. All methods analyzed the same 500 matrices for each combination of scenario and failure prevalence. Table 1 provides prediction coefficients used in four different evaluation scenarios; the additional four scenarios are obtained by combining the scenarios in table 1 with partly misclassified adherence data.

TABLE 1: Predictive Model Coefficients for Data Generation and Method Evaluation for our 4 scenarios; As Good As It Gets (AGAIG), Ignoring Adh, Ignoring PFS, and Reversing the Direction of Association Between our Predictors and log10 Viral Load

Coefficients	AGAIG	Ignoring Adh	Ignoring PFS	Reverse
Intercept	1	1	1	6
Adh	0.05	0	0.1	-0.05
PFS	1	3	0	-1
Interaction	0.05	0	0	-0.05

The As Good As It Gets (AGAIG) scenario uses the same prediction coefficients which generated the data when predicting individual HIV viral load. This represents the best-case scenario and should produce the best performance. In the other three scenarios we are misspecifying the model by either excluding important predictors or reversing the direction of association between our predictors and outcome. These are more realistic scenarios, because in practice we never know the true prediction model. By ignoring adherence and using only prior failure status, it makes sense to declare all non-prior failures as non-treatment failures and all prior failures as treatment failures. When ignoring PFS the prediction is solely dependent on adherence, so it is reasonable to assume the effect of adherence on treatment failure is

exaggerated. In the Reverse scenario the direction of association between both predictors and the outcome is reversed. This represents a possible worst-case scenario when using these predictive pooled testing methods in practice. Even choosing predictors with no association to HIV viral load should not produce as poor of performance as reversing the direction of association with actual predictors.

These four scenarios assume that all self-reported adherence data is accurate which is not likely in reality, so we also evaluate these four scenarios after permuting 40% of the adherence data. In each dataset of 1 million patients, we randomly selected 400,000 individuals and randomly permuted their adherence data. Therefore, the reported adherence may not be the adherence which contributed to the simulated viral load; representing the misclassification from self-reporting. We permuted the adherence data twice, and chose the higher of the two permutations to assign to those patients. Choosing the higher of the two values represents the upward bias in self-reporting. Evaluating the methods under all eight scenarios gives us a broad spectrum of realistic settings to answer to the question of how these methods perform.

Seeds were set to ensure that all methods evaluated the exact same 500 matrices, and the 50,000 patients were sampled without replacement from our datasets of 1 million. Different seeds were set when generating the datasets for the four failure prevalences.

3. Results

Table 2 provides mean estimates of relative efficiency, sensitivity and number of rounds in the ‘As Good As It Gets’ (AGAIG) scenario for the two current methods (MSS and Mini) which do not use covariate information and the three new methods (LRSOE, Linreg and Mini/Cov) that incorporate covariate information across four treatment failure prevalences of 1%, 5%, 10% and 20%.

TABLE 2: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'As Good As It Gets' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**77.2%**	**85.6%**	**1.8**	**67.0%**	**95.1%**	**3.2**
Linreg	78.2%	83.4%	2.0	68.2%	92.8%	4.9
Mini/Cov	83.4%	97.1%	4.4	66.7%	98.0%	9.0
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**52.0%**	**93.6%**	**5.2**	**23.8%**	**87.8%**	**7.7**
Linreg	50.4%	90.8%	9.1	21.1%	85.3%	15.3
Mini/Cov	48.6%	95.6%	10.7	27.1%	93.1%	11.0

The bolded, red estimates are the benchmark performance standards for the MSS method which we are attempting to improve on by incorporating covariate information. The LRSOE method shows the most promise for improving on the MSS method, so those estimates are also bolded.

The mean estimated performance for the LRSOE method is superior to the MSS method in every category at every treatment failure prevalence. Also, in this AGAIG scenario the Mini/Cov method appears to perform quite well in both efficiency and sensitivity while needing more rounds of testing. The Linreg method also appears to perform well with number of rounds of testing varying from one of the lowest at 1% prevalence to the highest at 20% failure prevalence.

Table 3 provides the same mean estimates as Table 2, but in the scenario where the direction of association between the predictors and log10 viral load is reversed.

TABLE 3: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Reverse Association' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.1%**	**83.2%**	**2.1**	**59.0%**	**92.8%**	**4.2**
Linreg	77.0%	80.7%	2.4	61.3%	91.6%	6.4
Mini/Cov	69.1%	92.7%	9.0	36.4%	97.7%	10.9
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**39.2%**	**92.4%**	**6.0**	**9.3%**	**88.6%**	**8.1**
Linreg	41.4%	91.5%	10.9	12.4%	87.1%	17.2
Mini/Cov	16.1%	98.8%	11.0	-3.7%	99.0%	11.0

Notice that the MSS and Mini methods have the exact same performance estimates as in the AGAIG scenario. Because these methods do not use covariate information we expect these estimates to be identical across all scenarios. The LRSOE method is still superior to the MSS method in both sensitivity and number of testing rounds at 5%, 10% and 20% failure prevalences while losing efficiency to the MSS method. The loss in efficiency compared with the MSS method also increases as the failure prevalence increases although the efficiency loss is not overwhelming (1%-8%). The Mini/Cov method which performed well in the AGAIG scenario actually loses efficiency at 20% failure prevalence in this Reverse scenario.

In Table 4 we again provide the mean performance estimates, this time for the Ignore PFS scenario after permuting 40% of the adherence data to reflect the misclassification and upward bias inherent with self-reported measurements.

TABLE 4: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Ignore PFS' Scenario After 40% Adherence Permutation

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.8%**	**83.9%**	**1.9**	**64.0%**	**93.4%**	**3.6**
Linreg	77.8%	81.4%	2.2	65.5%	91.3%	5.6
Mini/Cov	78.2%	96.2%	6.6	57.2%	97.2%	10.1
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**47.3%**	**92.2%**	**5.6**	**17.8%**	**87.6%**	**7.9**
Linreg	47.0%	91.6%	9.8	18.0%	86.3%	15.9
Mini/Cov	40.7%	96.6%	10.8	16.8%	94.8%	11.0

The efficiency and sensitivity of the LRSOE and MSS methods are very similar across all prevalences with the LRSOE method holding a distinct advantage over the MSS method in terms of number of testing rounds. The Mini/Cov method is again the leader in sensitivity while maintaining respectable efficiency, but at the cost of greater number of testing rounds.

Performance estimates for the scenarios not shown were as expected (complete results in appendix). When 40% of adherence data was permuted, the AGAIG, Ignore Adh and Reverse scenarios provided very similar estimates to their not-permuted counterparts. The Ignore Adh scenarios' performance estimates were very similar, if slightly worse, than the AGAIG scenarios across all failure prevalences.

4. Discussion

4.1 *Discussion of Results*

In Table 2 the improvements of the LRSOE method over the MSS method can be considered modest, however it is important to note that we are gaining in both efficiency and sensitivity while limiting the number of testing rounds. Even slight improvement in all 3 categories identifies a superior method. These results also show promise for the Mini/Cov method, and if number of testing rounds is inconsequential, one can easily argue that the Mini/Cov method is the preferred method. However, this is the AGAIG scenario and serves as a proof of concept for these methods rather than an honest evaluation of how these methods will perform in practice. In reality we never know the true model of association between our predictors and HIV viral load, but it may be possible to identify a useful prediction model for HIV viral load.

Using pre-existing data from a testing clinic, one can regress log10 viral load on covariates that are accessible without a blood test such as adherence and prior failure status. Doing so can identify the strongest accessible predictors, and estimate their effect on log10 HIV viral load. It may be optimal for each testing clinic to perform this independently as predictive models for HIV could depend on geographic and cultural differences. It may also be useful to compare predicted viral load values to individual test results to calibrate the prediction model, and repeat this process regularly. The benefits of covariate pooled testing methods depend on the ability to predict HIV viral load, and care should be taken to identify a strong prediction model when applying these methods.

Table 3 illustrates the potential pitfalls of using the Mini/Cov method in practice, because we actually lose efficiency at 20% failure prevalence if the predictive model is substantially incorrect. However, the LRSOE method still provides efficiency gains and a low number of testing rounds while maintaining high sensitivity in the Reverse scenario. In addition, with

efficiency gains less than 20% it may not be worth the added complexity of implementing matrix pooled testing methods when treatment failure prevalence is 20% or higher.

It may seem puzzling that a scenario which relies on predicting HIV viral load still performs well in the Reverse scenario. The LRSOE method performs well in this scenario, because it relies heavily on the row and column averages. It uses the viral load predictions only to guide the solution to the testing matrix; choosing where to begin the solution (detailed descriptions with R code in Appendix). If the prediction model wrongly predicts a patient's viral load to be high, as in the Reverse scenario, that sample's maximum viral load is still constrained by the combination of both row and column average. This is not the case in the Mini/Cov method, which is why that method performs badly in the Reverse scenario.

In reality we would not expect to reverse the direction of association with all of our predictors, so the results from the Reverse scenario can be viewed as worse than we're likely to encounter in practice. If sensitivity and testing turnaround-time and cost are the most important factors for a testing clinic, then the LRSOE method is slightly preferential to the MSS method. Even if efficiency is the prime concern, the LRSOE method does not lose a significant amount of efficiency over the MSS method.

In Table 4 we are ignoring PFS, so these estimates represent the scenario which has the most pronounced effect of permuting the adherence data. Even after permutation however, the LRSOE method appears preferential to the MSS method.

We have evaluated and presented performance estimates for relative efficiency, sensitivity and number of testing rounds in a variety of scenarios in order to identify methods which use covariate information and improve on the current best-performing method, MSS. We evaluated these methods in best-case and worst-case scenarios, given that we have predictors of

HIV viral load, while also accounting for the bias inherent in self-reported measurements. Based on these results, the LRSOE method appears preferential to the MSS method in scenarios likely to be encountered in practice, under the assumption that there exist accessible covariates predictive of HIV viral load. Even in the unlikely scenario where the direction of association with the predictors is reversed, the LRSOE method is similar to or preferential to the MSS method in terms of sensitivity and number of testing rounds while sacrificing a modest amount of efficiency.

4.2 *Study Limitations*

The most important limitations to this study are the assumptions that we have access to covariate information that is predictive of HIV viral load, and that we know the true data generating model. However, prior literature suggests that it is reasonable to assume that there does exist accessible covariates that are predictive of HIV viral load. Robbins et al. (2007) showed that treatment adherence and prior failure status were independent predictors of HIV viral load when also adjusting for other possible confounders. Also, even though we knew the data generating model, we purposely misclassified the model in order to obtain performance estimates in realistic scenarios. The purpose of this study is to answer the question of whether or not we can improve on known matrix pooled testing methods for detecting HIV treatment failure by incorporating informative covariate information. To answer this, we must assume that informative covariate information exists, which we did.

Other limitations of this study relate to the design of matrix testing and data simulation. We only evaluated methods using matrices of size 10 x 10 with 4 different failure prevalences. When generating data, we only used 0.5 on the log10 scale as the standard deviation of person-to-person variability. Increasing this standard deviation would reflect a weaker association

between our predictors and log10 viral load. We evaluated our methods using only 1 model which included a continuous, binary and interaction term. This paper serves as a proof of concept that we can improve on the MSS method by incorporating informative covariate information, and should not be taken as the optimal matrix pooled testing design under these conditions.

Further study is needed to optimize the strategies outlined in this paper by exploring more scenarios to focus on the optimal performance design parameters for each of these methods which may vary from region to region and testing method to testing method. Also needed is the evaluation of these methods in practice; combining a prediction dataset to calibrate a prediction model with patient data to be tested.

References

Behets F., Bertozzi S., Kasali M., et al. (1990). Successful use of pooled sera to determine HIV-1 seroprevalence in Zaire with development of cost-efficiency models. *AIDS* **4(8)**:737–741.

Brookmeyer R. (1999). Analysis of multistage pooling studies of biological specimens for estimating disease incidence and prevalence. *Biometrics* **55(2)**:608–612.

Busch M.P., Glynn S.A., Stramer S.L., et al. (2005). A new strategy for estimating risks of transfusion transmitted viral infections based on rates of detection of recently infected donors. *Transfusion* **45(2)**:254–264.

Cahoon-Young B., Chandler A., Livermore T., et al. (1989). Sensitivity and specificity of pooled versus individual sera in a human immunodeficiency virus antibody prevalence study. *J Clin Microbiol* **27(8)**:1893–1895.

Cohen M.S., Chen Y.Q., McCauley M., et al. (2011). Prevention of HIV-1 Infection with Early Antiretroviral Therapy. *N Engl J Med* **365(8):** 493-505.

Dorfman R. (1943). The detection of defective members of large populations. *Annals of Mathematical Statistics* **14(4)**:436–440.

Gastwirth J.L., Hammick P.A. (1989). Estimation of the prevalence of a rare disease, preserving the anonymity of the subjects by group-testing—application to Estimating the prevalence of AIDS antibodies in blood-donors. *Journal of Statistical Planning and Inference* **22(1)**:15–27.

Hammick P.A., Gastwirth J.L. (1994). Group-testing for sensitive characteristics: extension to higher prevalence levels. *International Statistical Review* **62(3)**:319–331.

Hanscom B.S. (2014). Biostatistical Methods for HIV Monitoring and Prevention. Dissertation, University of Washington.

Hanscom B.S., May S., Hughes J.P. (2014). Efficiently Identifying Failures using Quantitative Tests, Matrix Pooling and the EM-Algorithm. *BePress, UW Biostatistics Working Paper Series*.

Kim H.Y., Hudgens M.G., Dreyfuss J.M., et al. (2007). Comparison of group testing algorithms for case identification in the presence of test error. *Biometrics* **63(4)**: 1152 – 1163.

Kline R.L., Brothers T.A., Brookmeyer R., et al. (1989). Evaluation of human immunodeficiency virus seroprevalence in population surveys using pooled sera. *J Clin Microbiol* **27(7)**:1449–1452.

May S., Gamst A., Haubrich R., Benson C., Smith D.M. (2010). Pooled Nucleic Acid Testing to Identify Antiretroviral Treatment Failure During HIV Infection. *J Acquir Immune Defic Syndr* **53(2)**: 194-201.

Patterson K.B., Leone P.A., Fiscus S.A., et al. (2007). Frequent detection of acute HIV infection in pregnant women. *AIDS* **21(17)**:2303–2308.

Pilcher C.D., Fiscus S.A., Nguyen T.Q., et al. (2005). Detection of acute infections during HIV testing in North Carolina. *N Engl J Med* **352**:1873–1883.

Pilcher C.D., McPherson J.T., Leone P.A., et al. (2002). Real-time, universal screening for acute HIV infection in a routine HIV counseling and testing population. *JAMA* **288(2)**:216–221.

Quinn T.C., Brookmeyer R., Kline R., et al. (2000). Feasibility of pooling sera for HIV-1 viral RNA to diagnose acute primary HIV-1 infection and estimate HIV incidence. *AIDS* **14(17)**:2751–2757.

Robbins G.K., Daniels B., Zheng H., et al. (2007). Predictors of Antiretroviral Treatment Failure in an Urban HIV Clinic. *J Acquir Immune Defic Syndr* **44(1)**: 30-37.

Tu X.M., Litvak E., Pagano M. (1995). On the Informativeness and accuracy of pooled testing in estimating prevalence of a rare disease—application to HIV screening. *Biometrika* **82(2)**:287–297.

Westreich D.J., Hudgens M.G., Fiscus S.A., et al. (2008). Optimizing screening for acute HIV infection with pooled nucleic acid amplification tests. *J Clin Microbiol* **46(5)**:1785–1792.

Appendix

A1. *Complete Results*

TABLE A1: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'As Good As It Gets' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**77.2%**	**85.6%**	**1.8**	**67.0%**	**95.1%**	**3.2**
Linreg	78.2%	83.4%	2.0	68.2%	92.8%	4.9
Mini/Cov	83.4%	97.1%	4.4	66.7%	98.0%	9.0
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**52.0%**	**93.6%**	**5.2**	**23.8%**	**87.8%**	**7.7**
Linreg	50.4%	90.8%	9.1	21.1%	85.3%	15.3
Mini/Cov	48.6%	95.6%	10.7	27.1%	93.1%	11.0

TABLE A2: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'As Good As It Gets' Scenario After 40% Adherence Permutation

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**77.1%**	**85.3%**	**1.8**	**66.2%**	**94.7%**	**3.3**
Linreg	78.1%	83.2%	2.0	67.2%	91.4%	5.1
Mini/Cov	83.2%	97.1%	4.5	64.7%	97.8%	9.3
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**49.6%**	**92.9%**	**5.4**	**22.0%**	**87.4%**	**7.7**
Linreg	48.8%	91.0%	9.4	20.2%	85.5%	15.5

Mini/Cov	44.5%	95.9%	10.8	23.7%	93.5%	11.0

TABLE A3: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Reverse Association' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.1%**	**83.2%**	**2.1**	**59.0%**	**92.8%**	**4.2**
Linreg	77.0%	80.7%	2.4	61.3%	91.6%	6.4
Mini/Cov	69.1%	92.7%	9.0	36.4%	97.7%	10.9
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**39.2%**	**92.4%**	**6.0**	**9.3%**	**88.6%**	**8.1**
Linreg	41.4%	91.5%	10.9	12.4%	87.1%	17.2
Mini/Cov	16.1%	98.8%	11.0	-3.7%	99.0%	11.0

TABLE A4: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Reverse Association' Scenario After 40% Adherence Permutation

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.1%**	**83.2%**	**2.1**	**59.0%**	**92.8%**	**4.2**
Linreg	77.0%	80.7%	2.4	61.3%	91.6%	6.4
Mini/Cov	69.1%	92.7%	9.0	36.4%	97.7%	10.9
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**39.2%**	**92.4%**	**6.0**	**9.3%**	**88.6%**	**8.1**
Linreg	41.4%	91.5%	10.9	12.4%	87.1%	17.2
Mini/Cov	16.1%	98.8%	11.0	-3.7%	99.0%	11.0

TABLE A5: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Ignore PFS' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.9%**	**83.8%**	**1.9**	**65.2%**	**93.7%**	**3.5**
Linreg	77.9%	81.8%	2.1	66.7%	91.3%	5.3
Mini/Cov	78.8%	96.4%	6.4	60.6%	97.6%	9.8
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**49.6%**	**93.2%**	**5.3**	**20.2%**	**87.7%**	**7.8**
Linreg	48.7%	91.6%	9.5	19.2%	86.2%	15.8
Mini/Cov	45.0%	96.7%	10.8	20.4%	94.7%	11.0

TABLE A6: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Ignore PFS' Scenario After 40% Adherence Permutation

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**76.1%**	**83.7%**	**2.1**	**59.2%**	**92.6%**	**4.1**
Linreg	77.1%	80.7%	2.4	61.7%	91.2%	6.3
Mini/Cov	69.4%	92.9%	8.9	38.3%	97.9%	10.8
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**39.8%**	**92.7%**	**6.0**	**9.8%**	**88.6%**	**8.1**
Linreg	42.0%	91.4%	10.8	12.8%	86.9%	17.0
Mini/Cov	19.3%	98.6%	11.0	-1.5%	98.5%	11.0

TABLE A7: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Ignore Adherence' Scenario

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**77.1%**	**86.1%**	**1.8**	**65.8%**	**94.0%**	**3.4**
Linreg	78.1%	83.5%	2.0	66.9%	92.5%	5.3
Mini/Cov	82.6%	95.5%	4.7	61.9%	98.7%	9.6
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**47.9%**	**92.0%**	**5.5**	**20.4%**	**87.4%**	**7.8**
Linreg	47.4%	91.4%	9.8	19.3%	86.4%	15.9
Mini/Cov	41.4%	96.9%	10.9	18.5%	94.6%	11.0

TABLE A8: Comparison of Mean Relative Efficiency, Sensitivity and Number of Rounds Based on 500 Replications Across Treatment Failure Prevalences of 1%, 5%, 10% and 20% in the 'Ignore Adherence' Scenario After 40% Adherence Permutation

	1%			5%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**76.9%**	**83.7%**	**1.9**	**64.5%**	**91.2%**	**4.8**
Mini	76.4%	96.8%	7.7	51.4%	97.4%	10.6
LRSOE	**77.1%**	**86.1%**	**1.8**	**65.8%**	**94.0%**	**3.4**
Linreg	78.1%	83.5%	2.0	66.9%	92.5%	5.3
Mini/Cov	82.6%	95.5%	4.7	61.9%	98.7%	9.6
	10%			20%		
	Eff.	Sens.	Rds.	Eff.	Sens.	Rds.
MSS	**46.3%**	**89.8%**	**8.5**	**17.3%**	**85.0%**	**14.3**
Mini	31.2%	96.9%	11.0	10.1%	95.9%	11.0
LRSOE	**47.9%**	**92.0%**	**5.5**	**20.4%**	**87.4%**	**7.8**
Linreg	47.4%	91.4%	9.8	19.3%	86.4%	15.9
Mini/Cov	41.4%	96.9%	10.9	18.5%	94.6%	11.0

A2. *Data Simulation Detail*

A2.1 *Simulating Realistically Skewed Viral Load Values*

Below are the descriptions and R code for the functions that simulated realistic viral load values based on the research by May et al. (2010).

```
# This function simulates viral load (VL) values under 500 copies/mL
# n is the number of values to simulate
Under500VL <- function(n){
  a <- round(.85*n)
  b <- round(.05*n)
  c <- n - a - b
  # 85% ~ U(0,50); 5% ~ U(50,100); 10% ~ U(100,500)
  y <- c(runif(a,0,50), runif(b,50,100),runif(c,100,500))
  # returns a vector of viral load values under 500 copies/mL
  return(y)}

# Simulates VL values above 500 copies/mL
# n is the number of values to simulate
Over500VL <- function(n){
  a <- round(.93*n)
  b <- n - a
  # 93% ~ Gamma(1.6, 0.5); 7% ~ Gamma(3.18, 0.5); both shifted 2.7 right
  y <- c(replicate(a, 10^(2.7 + rgamma(1,shape = 1.6, scale = 0.5))),
         replicate(b, 10^(2.7 + rgamma(1,shape = 3.18, scale = 0.5))))
  # returns a vector of viral load values over 500 copies/mL
  return(y)}
```

Notice that combined with a failure prevalence over 500 copies/mL, it is possible to simulate a set of realistically skewed viral load values with the preferred prevalence over 500 copies/mL. However, we use a treatment failure cutoff of 1000 copies/mL. Therefore, we need some way of converting a failure prevalence at a cutoff of 1000 into a failure prevalence at a cutoff of 500 in order to maintain the same mixed viral load distribution. The next function does this.

```
# Converts a failure prevalence at any cutoff into a failure prevalence above
500
prevover500 <- function(prevfail, cutoff){
  # This uses the cumulative distribution functions of viral load values to
  # convert prevalence at any cutoff into prevalence over 500.
  # Note:  this depends on where the original failure cutoff is.
  if (50 <= cutoff & cutoff < 100){
    prevover500 = (prevfail - .10 - (.05*(100-cutoff)/50))/(1 - .1 - (.05*(10
0-cutoff)/50))}
  else if (100 <= cutoff & cutoff < 500){
    prevover500 = (prevfail - (.10*(500-cutoff)/400))/(1 - (.10*(500-cutoff)/
400))}
  else if (cutoff == 500){
    prevover500 = prevfail}
```

```
  else if (cutoff > 500){
    prevover500 = prevfail/((.93*(1 - pgamma(log10(cutoff)-2.7, shape = 1.6,
scale = 0.5)) +
                  .07*(1 - pgamma(log10(cutoff)-2.7, shape = 3.18, scale = 0.
5))))}
  else {prevover500 = "error"}
  # returns an euivalent failure prevalence for a cutoff of 500
  return(prevover500)
}
```

Now that we can convert any failure prevalence into an equivalent failure prevalence above 500, we are ready to simulate viral load values at any prevalence at any cutoff.

```
# Creates a vector of realistically skewed viral load values
# n is the number of VL's to simulate
# prevover500 is the prevalence above the cutoff (assumed 500)
# cutoff is the cutoff defining treatment failure
# prevfail is a boolean.  If true, the function will interpret
# prevover500 as the prevalence over the cutoff
createpop <- function(n, prevover500, cutoff, prevfail=FALSE){
  if (prevfail==FALSE){
    prevover500 = prevover500}
  # If prevfail=TRUE, this converts the prevover500
  # into an actual prevalence over 500 copies/mL
  else {prevover500 <- prevover500(prevover500, cutoff)}
  # Calculates the number over 500
  a <- round(n*prevover500)
  # Calculates the number under 500
  b <- n - a
  # Simulates VL's based on mixed distribution by May et al. (2020)
  over500 <- Over500VL(a)
  under500 <- Under500VL(b)
  pop <- c(over500, under500)
  # reutrns vector of VL's
  return(pop)
}
```

The output from createpop represent realistically skewed viral load values as seen in practice by May et al. (2010). We use these viral load values to compare to our viral load values generated from our covariate model using our shiny application described in the next section.

A2.2 *Data-Fitting Shiny Application*

The challenge here was simulating viral load values from a covariate model, and obtaining viral load values similar to those seen in practice. Therefore, we wrote a shiny application which generates viral load values from both createpop and our covariate model, makes histograms, and visually shows the overlap between the two. The shiny application allows the user to change the parameters of the covariate model to attempt to match the distribution of VL's to those seen in practice. This is how we obtained distributional parameters for the adherence variable. Below are the 3 files necessary for running the application. In order to run the application, you need a new

folder named "Blank" in your R working directory. Copy the 3 files into that folder, install the shiny package from R, and type: "runApp("Blank")".

```
# This code needs to be in a separate R file called server.R
library(shiny)
source("shinysource.R")

shinyServer(function(input, output){

    output$plot2 <- renderPlot({
      set.seed(input$seed)

      plot.pop(n=input$popsize, prev.over.cutoff=input$prev, cutoff=input$cut
off,
               shape1=input$shape1, shape2=input$shape2, pf.prev=input$pf, b0
=input$b0,
               b1=input$b1, b2=input$b2, b3=input$b3, sd=input$sd)
    })
})

# This code also needs to be in a separate file, but called ui.R
library(shiny)
source("shinysource.R")

shinyUI(fluidPage(
  titlePanel("Parametric Population Fit"),

  sidebarLayout(
    sidebarPanel(
                 numericInput("seed", label = "Seed", value=12, min=1),
                 numericInput("popsize", label = "Pop. Size", value=1000, min
=1000),
                 numericInput("b0", label = "B0", value=1, min=-50),
                 numericInput("b1", label = "B1", value=.05, min=-50),
                 numericInput("b2", label = "B2", value=1, min=-50),
                 numericInput("b3", label = "B3", value=.05, min=-50),
                 numericInput("sd", label = "Standard Deviation", value=.5, m
in=0.01),
                 sliderInput("cutoff", label = "Failure Cutoff", min=200, max
=2500, value=1000, step=50),
                 sliderInput("prev", label = "Prevalence Over Cutoff", min=.0
1, max=1, value=0.1),
                 sliderInput("pf", label = "Prior Failure Prevalence", min=.0
1, max=1, value=0.25),
                 numericInput("shape1", label = "Beta Shape 1", min=0.1, valu
e=5),
                 numericInput("shape2", label = "Beta Shape 2", min=0.1, valu
e=0.5)
                 ),
```

```
    mainPanel(
            h1("Population vs Parametric Fit: Combined curves"),
            plotOutput("plot2")
    )
  )
  ))

# Finally, also in a separate R file, call this one shinysource.R

Under500VL <- function(n){
  a <- round(.85*n)
  b <- round(.05*n)
  c <- n - a - b
  y <- c(runif(a,0,50), runif(b,50,100),runif(c,100,500))
  return(y)
}

Over500VL <- function(n){
  a <- round(.93*n)
  b <- n - a
  y <- c(replicate(a, 10^(2.7 + rgamma(1,shape = 1.6, scale = 0.5))),
         replicate(b, 10^(2.7 + rgamma(1,shape = 3.18, scale = 0.5))))
  return(y)
}

prevover500 <- function(prevfail, cutoff){
  if (50 <= cutoff & cutoff < 100){
    prevover500 = (prevfail - .10 - (.05*(100-cutoff)/50))/(1 - .1 - (.05*(10
0-cutoff)/50))}
  else if (100 <= cutoff & cutoff < 500){
    prevover500 = (prevfail - (.10*(500-cutoff)/400))/(1 - (.10*(500-cutoff)/
400))}
  else if (cutoff == 500){
    prevover500 = prevfail}
  else if (cutoff > 500){
    prevover500 = prevfail/((.93*(1 - pgamma(log10(cutoff)-2.7, shape = 1.6,
scale = 0.5)) +
                               .07*(1 - pgamma(log10(cutoff)-2.7, shape = 3.1
8, scale = 0.5))))}
  else {prevover500 = "error"}
  return(prevover500)
}

createpop <- function(n, prevover500, cutoff, prevfail=TRUE){
  if (prevfail==FALSE){
    prevover500 = prevover500}
  else {prevover500 <- prevover500(prevover500, cutoff)}
  a <- round(n*prevover500)
  b <- n - a
```

```
  over500 <- Over500VL(a)
  under500 <- Under500VL(b)
  pop <- c(over500, under500)
  return(pop)
}

# Simulates a dataframe with adherence, PFS and viral load for n patients
# based on the covariate model, described in detai later
VL.data.adh <- function(n, shape1, shape2, b0, b1, b2, b3, pf.prev, sd){
  pf <- rbinom(n=n, size=1, prob=pf.prev)
  adhere <- (100 - 100*rbeta(n=n, shape1 = shape1, shape2=shape2))
  var <- rnorm(n=n, mean=0, sd = sd)
  log.VL <- b0 + b1*adhere + b2*pf + b3*adhere*pf + var
  VL <- 10^log.VL
  data <- cbind(VL, adhere, pf)
  return(data.frame(data))
}

# Plots the histograms
plot.pop <- function(n, prev.over.cutoff, cutoff, shape1, shape2, pf.prev, b0
, b1, b2, b3, sd){
  data <- createpop(n, prev.over.cutoff, cutoff, prevfail=TRUE)
  Logdata <- log10(data)
  modeldata <- VL.data.adh(n=n, shape1=shape1, shape2=shape2, b0=b0, b1=b1, b
2=b2, b3=b3, pf.prev=pf.prev, sd=sd)
  modelVL <- log10(modeldata$VL)
  hist(Logdata, freq=FALSE, col=rgb(1,0,0,0.5), xlim = c(0,10), breaks=21,
       xlab="Log10(VL); Red is Pop VL, Blue is Model Fit", main="Log10 VL")
  hist(modelVL, col=rgb(0,0,1,0.5), freq=FALSE, add=TRUE, breaks=21)
  box()
}
```

End of Shiny Application Code

A2.3 *Simulating Final Data*

Using our shiny application we were able to obtain parameter values which provide us with the distribution of viral load values likely seen in practice at the desired treatment failure prevalence. In order to simulate as close to reality as possible, we chose model coefficient values based on previous literature and scientific knowledge. Therefore, the coefficients remain fixed durng our data fitting with the shiny application. We also fixed 25% previous failure status (PFS) prevalence, failure cutoff of 1000 copies/mL, and standard deviation for person-to-person variablility at 0.5 on the log10 scale. The only parameters we changed were the distributional parameters for the adherence variable. This seems reasonable as previous literature suggests that adherence is the strongest predictor of HIV viral load.

Now that we have the desired parameters for our covariate model, we are ready to generate data.

```
# Simulates a dataframe with adherence, PFS and viral load for n patients
# based on the covariate model, described in detai later
```

```
# Adherence data ~ Beta(shape1, shape2)
# b0 - b4 are the coefficients of our covariate model
# pf.prev is the previous failure status
# sd is the standard deviation for person-to-person variation
VL.data.adh <- function(n, shape1, shape2, b0, b1, b2, b3, pf.prev, sd){
  # simulates n previous failure statuses usng independent Bernoullis
  # where p is the input for pf.prev
  pf <- rbinom(n=n, size=1, prob=pf.prev)
  # Simulates adherence data for n patients based on distr. pars.
  adhere <- (100 - 100*rbeta(n=n, shape1 = shape1, shape2=shape2))
  # simulated person to person variablilty
  var <- rnorm(n=n, mean=0, sd = sd)
  # generates log10 VL from our covariate model
  log.VL <- b0 + b1*adhere + b2*pf + b3*adhere*pf + var
  # converts to actual viral load values
  VL <- 10^log.VL
  data <- cbind(VL, adhere, pf)
  # reutrns a data frame with  VL, adherence and PFS data
  # for each patient
  return(data.frame(data))
}
```

Using the above functions, we generated complete datasets of 1 million patients at each of 4 treatment failure prevalences; 1%, 5%, 10% and 20%. Although failure prevalences were not exact, all were within 0.5% of the desired prevalence. The code we used to generate these data is below.

```
# Generates 1 million patient data at 1% failure prevalence over 1000 copies/
mL
set.seed(8)
data125 <- VL.data.adh(n=1000000, shape1=8, shape2 = 0.05, b0=1, b1=.05, b2=1
, b3=.05, pf.prev=.25, sd=0.5)
write.table(data125, file="data125.shapes.8.0.0.05betas1..05.1..05sd.5")

# Generates 1 million patient data at 5% failure prevalence over 1000 copies/
mL
set.seed(11)
data525 <- VL.data.adh(n=1000000, shape1=7, shape2 = 0.3, b0=1, b1=.05, b2=1,
b3=.05, pf.prev=.25, sd=0.5)
write.table(data525, file="data525.shapes.7.0.0.3betas1..05.1..05sd.5")

# Generates 1 million patient data at 10% failure prevalence over 1000 copies
/mL
set.seed(12)
data1025 <- VL.data.adh(n=1000000, shape1=4, shape2 = 0.33, b0=1, b1=.05, b2=
1, b3=.05, pf.prev=.25, sd=0.5)
write.table(data1025, file="data1025.shapes.4.0.0.33betas1..05.1..05sd.5")

# Generates 1 million patient data at 20% failure prevalence over 1000 copies
/mL
```

```
set.seed(15)
data2025 <- VL.data.adh(n=1000000, shape1=6, shape2 = 1.05, b0=1, b1=.05, b2=
1, b3=.05, pf.prev=.25, sd=0.5)
write.table(data2025, file="data2025.shapes.6.0.1.05betas1..05.1..05sd.5")
```

Next we truncate the VL's at a maximum of 2,000,000 copie/mL to represent the upper limit of detection. The trunc number at the bottom of each section of code is the number of truncated VL's out of 1 million.

```
# Truncation code for 1% failure prevalence
data125 <- read.table("data125.shapes.8.0.0.05betas1..05.1..05sd.5")
# Truncates all VL's at 2,000,000
trunc125 <- 0
for (i in 1:length(data125[,1])){
  if (data125$VL[i] > 2000000){
    data125$VL[i] <- 2000000
    trunc125 <- trunc125 + 1
  }
}
# Overwrites the old data
write.table(data125, file="data125.shapes.8.0.0.05betas1..05.1..05sd.5")
# This is the number of VL's that were truncated out of 1 million
# trunc125 = 49

# Truncation code for 5% failure prevalence
data525 <- read.table("data525.shapes.7.0.0.3betas1..05.1..05sd.5")

trunc525 <- 0
for (i in 1:length(data525[,1])){
  if (data525$VL[i] > 2000000){
    data525$VL[i] <- 2000000
    trunc525 <- trunc525 + 1
  }
}
write.table(data525, file="data525.shapes.7.0.0.3betas1..05.1..05sd.5")
# trunc525 = 844

# Truncation code for 10% failure prevalence
data1025 <- read.table("data1025.shapes.4.0.0.33betas1..05.1..05sd.5")

trunc1025 <- 0
for (i in 1:length(data1025[,1])){
  if (data1025$VL[i] > 2000000){
    data1025$VL[i] <- 2000000
    trunc1025 <- trunc1025 + 1
  }
}
write.table(data1025, file="data1025.shapes.4.0.0.33betas1..05.1..05sd.5")
## trunc1025 = 6105
```

```
# Truncation code for 20% failure prevalence
data2025 <- read.table("data2025.shapes.6.0.1.05betas1..05.1..05sd.5")

trunc2025 <- 0
for (i in 1:length(data2025[,1])){
  if (data2025$VL[i] > 2000000){
    data2025$VL[i] <- 2000000
    trunc2025 <- trunc2025 + 1
  }
}
write.table(data2025, file="data2025.shapes.6.0.1.05betas1..05.1..05sd.5")
## trunc2025 = 10450
```

Here we provide the code used for permuting our adherence data.

```
# This code permutes 40% of adherence data for each prevalence
data2025 <- read.table("data2025.shapes.6.0.1.05betas1..05.1..05sd.5")
# takes the first 40% of adherence values and permutes them twice
data <- data2025[1:400000,]
adhere <- data[,2]
adhere2 <- sample(adhere, size=400000)
adhere3 <- sample(adhere, size=400000)
# chooses the higher of the two permuted values
adhere4 <- NULL
for(i in 1:length(adhere2)){
  adhere4[i] <- max(adhere2[i], adhere3[i])
}
# assigns the new adherence value
for(i in 1:length(adhere4)){
  data2025$adhere[i] <- adhere4[i]
}
# writes the data into a different file name to keep datasets separate
write.table(data2025, file="data2025perm40.shapes.6.0.1.05betas1..05.1..05sd.
5")


data1025 <- read.table("data1025.shapes.4.0.0.33betas1..05.1..05sd.5")
data <- data1025[1:400000,]
adhere <- data[,2]
adhere2 <- sample(adhere, size=400000)
adhere3 <- sample(adhere, size=400000)
adhere4 <- NULL
for(i in 1:length(adhere2)){
  adhere4[i] <- max(adhere2[i], adhere3[i])
}
for(i in 1:length(adhere4)){
  data1025$adhere[i] <- adhere4[i]
}
write.table(data1025, file="data1025perm40.shapes.4.0.0.33betas1..05.1..05sd.
5")
```

```
data525 <- read.table("data525.shapes.7.0.0.3betas1..05.1..05sd.5")
data <- data525[1:400000,]
adhere <- data[,2]
adhere2 <- sample(adhere, size=400000)
adhere3 <- sample(adhere, size=400000)
adhere4 <- NULL
for(i in 1:length(adhere2)){
  adhere4[i] <- max(adhere2[i], adhere3[i])
}
for(i in 1:length(adhere4)){
  data525$adhere[i] <- adhere4[i]
}
write.table(data525, file="data525perm40.shapes.7.0.0.3betas1..05.1..05sd.5")


data125 <- read.table("data125.shapes.8.0.0.05betas1..05.1..05sd.5")
data <- data125[1:400000,]
adhere <- data[,2]
adhere2 <- sample(adhere, size=400000)
adhere3 <- sample(adhere, size=400000)
adhere4 <- NULL
for(i in 1:length(adhere2)){
  adhere4[i] <- max(adhere2[i], adhere3[i])
}
for(i in 1:length(adhere4)){
  data125$adhere[i] <- adhere4[i]
}
write.table(data125, file="data125perm40.shapes.8.0.0.05betas1..05.1..05sd.5"
)
```

This completes our data generation, and provides us with all of the data we will need for our simulations. This is the actual code used in our simulations, so another researcher should be able to reproduce our results exactly.

A3. *Evaluation Simulation*

In this section we discuss how we evalauted each pooled testing method, and provide code which should allow researchers to reproduce our results once they have reproduced the data from the above code.

A3.1 *Pooled Testing Set Up*

Most of the functions in this sub-section are used by all pooled testing methods, and provide a basis to better understand the functions which drive the methods. The first is a function that creates our testing matrix.

```
# function which makes one testable matrix from the population
# pop is a vector of VL's
```

```
# matsize is the size of the square matrix to test
# SE is the standard deviation of measurement error
one.matrix <- function(pop, matsize, SE){
  # blank matrix
  m <- matrix(nrow=matsize, ncol=matsize)
  # randomly selects samples from the population of VL's
  # without replacement
  for (i in 1:matsize){
    for(j in 1:matsize){
      m[i,j] = sample(pop,1,replace=FALSE)
    }}
  # creates a data frame of true VL's
  truth <- data.frame(m)
  error <- data.frame(m)
  # creates a data frame of observable VL's
  # I.e., VL's with measurement error
  for (i in 1:matsize){
    for(j in 1:matsize){
      error[i,j] = error[i,j]*(10^(rnorm(1, sd=SE)))}}
  # Creates a data frame combining the true and observable VL's
  x <- data.frame(c(truth,error))
  rowst <- replicate(matsize, 0)
  # Creates the row and column averages, also with
  # measurement error applied
  for (i in 1:matsize){rowst[i] = sum(x[i,1:matsize]/matsize)}
  rows <- rowst*(10^(rnorm(1, sd=SE)))
  colst <- replicate(matsize, 0)
  for (i in 1:matsize){colst[i] = sum(x[1:matsize,i]/matsize)}
  cols <- colst*(10^(rnorm(1, sd=SE)))
  # returns a matrix where the first n columns are the true VL's,
  # the next n columns are the observable values
  # the final 2 rows are the row and column averages
  # with measurement error, respectively
  return(cbind(x,rows,cols))
}
```

Notice here that we end up with a matrix with 2n + 2 columns. The first n columns are the true VL's for each patient in our matrix, the next n colums are the observable VL's for those patients, and the last 2 are the row and column averages with measurement error, respectively. This is necessary for tracking which patients we actually test and which patients would've tested above the cutoff in order to accurately measure sensitivity. Also, observable VL's are needed, because we can never observe the true VL due to measurement error.

The next function determines which row and column averages are below *t/n*, and zeros out those rows and columns to reflect classifying those individuals as not experiencing treatment failure. This function also assigns VL equal to the row/column average to each patient classified in this way. This is necessary for altering the column or row averages to account for the newly classified patients. It is imprtant to note for our sensitivity calculation that we do distinguish between classsifying a patient and actually testing them. More on this later.

```
# zeroes out all rows and columns below t/n
# y is our testing matrix
zerocolrow <- function(y, matsize, cutoff, lowlimit){

  # checks each row to see if its below the cutoff/matsize
  for(i in 1:matsize){
    if (y$rows[i] < cutoff/matsize){
      # this chunk zeros out the row total
      num <- y$rows[i]
      y$rows[i] = 0
      # alters col totals based on assigning equal VL value to each box in ro
w which
      # has not already been tested
      for (k in 1:matsize){
        if (y[i, (matsize+k)] == 0){}
        else{
          y$cols[k] <- y$cols[k] - num/matsize
          # zeros out the patients box in the testing matrix to signify that
          # the patient has been classified
          y[i,k] = 0
          y[[i, (matsize+k)]] = 0
        }
      }
    }
    else{}}
  # this is the same as the above code, but for the columns
  for(i in 1:matsize){
    if (y$cols[i] < cutoff/matsize){
      num <- y$cols[i]
      y$cols[i] = 0
      for (k in 1:matsize){
        if (y[k, (matsize+i)] == 0){}
        else{
          y$rows[k] <- y$rows[k] - num/matsize
          y[k,i] = 0
          y[[k, (matsize+i)]] = 0
        }
      }
    }
    else{}}
  # reutrns the testing matrix with the updated info
  return(y)
}
```

The next pooled testing function checks all patients in each row and column to determine if every patient in a row/column have been classified. If so, this function zeroes out that row/column average. This function is essential for avoiding infinite loops. Due to measurement error, it is possible for all patients in a row to be tested, and yet the row average still be greater than *t/n*. The below function fixes this issue.

```
# checks if all samples in a row or column have been tested
# If so, this zeroes out the row/column average
checkrowcol <- function(y, n){
  for (i in 1:n){
    if(sum(y[i,1:n]) == 0){y$rows[i] = 0}
    else{}
    if(sum(y[1:n,i]) == 0){y$cols[i] = 0}
    else{}
  }
  return(y)
}
```

This next function calculates the sensitivity of a classified (fully reduced/tested) matrix. We define sensitivity, or observable sensitivity, as the percent of failures we identify out of the failures we would have identified had we tested every patient. To do this we pass in 2 matrices. One is identical to our initial testing matrix. The next is a partially reduced testing matrix. This matrix has a zero in the true VL columns for every patient whom we actually tested. We can compare who we tested with the observable values and the true values using both matrices to calculate both true and observable sensitivity. True sensitivity is the percent of failures we identify out of everyone who was truly a failure, i.e., their true, unobservable VL is above *t/n*.

```
# Calculates sensitivity; both true sensitivity and observable sensitivity
# y is a reduced matrix with a zero in the true VL columns for every
# patient who was tested
# z is the original testing matrix
sens <- function(y, z, matsize, cutoff){
  a <- 0
  b <- 0
  n <- matsize +1
  m <- matsize*2
  # this loop checks for each true failure, how many do we conclude are failu
res
  # (True sens.)
  for (i in 1:matsize){
    for (j in 1:matsize){
      if (z[i,j]>cutoff){
        if (y[i,j]==0 & y[[i,(matsize + j)]]>cutoff){
          a <- a+1}
        else{a <- a}}
      else{}}}
  # this loops checks for each observable failure, how many do we conclude ar
e failures
  # (Observable sens.)
  for (i in 1:matsize){
    for (j in 1:matsize){
      if (z[[i,(matsize + j)]]>cutoff){
        if (y[i,j]==0 & y[[i,(matsize + j)]]>cutoff){
          b <- b+1}
        else{b <- b}}
      else{}}}
```

```
  truth <- z[,1:matsize]
  error <- z[,n:m]
  # Calculates percents
  true.sens <- a/length(truth[truth>cutoff])
  error.sens <- b/length(error[error>cutoff])
  # returns true sense and observable sens (error sens)
  return (c(true.sens, error.sens))
}
```

A3.2 *Method Description*

Below is a detailed description of each pooling method evalauted in this paper. Included is a list of steps that each method follows when choosing which and how many samples to test as well as the accompanying R code. This is intended for those who wish to reproduce these results using the methods described in this paper.

The heart of each method are the reduce.mat functions. These functions take in a new testing matrix with the row and column averages already tested/computed, apply the method to classify all patients as either treatment failure or non-treatment failures, and return the results for that matrix. Note that all matrices evaluated in this study were 10 x 10 in size, however the code allows the user to change the size of the matrix.

A3.2.1 *Modified Simple Search (MSS)*

Below are steps for classifying a matrix of samples using the MSS method; reduce.mat.smt.

- Tracks the number of tests and rounds conducted. Because the matrix is passed in with all row and column averages computed, we start with the matrix size times 2 number of tests and 1 round of testing.
- Tracks prevalence of failure metrics for diagnostic purposes.
- Determines if any row/column averages are less than *t/n*, and if so classify all of those individuals in those rows/columns as non-treatment failures.
- Start of while loop: Check if all row averages AND all column averages are greater than *t/n*.
  - *Note: This is the loop which determines when all patients are classified, and thus when to stop testing. Notice that this implies that we stop testing when EITHER all row avergaes or all column averages are less than* t/n. *This is due to the fact that we classify a sample as a non-treatment failure if either their row or column average is less than* t/n.
  - Calculates/recalculates/tracks the number of rows and the number of columns which can have a treatment failure, i.e., their value is above *t/n*. Also tracks the minimum of these two values.
  - Creates an *n x n* matrix of the sums (called sums) of every combination of row and column averages with the indices corresponding to the row and column numbers in the sum. This gives a matrix where every element is the sum of the row and column averages corresponding to each individual in the matrix.
  - Orders the values in the matrix of sums from highest to lowest, and tracks the indices corresponding to those sums in data frame d. Also tracks the number of samples left to classify.

  - The next section of "if", "else" statements chooses how many patients to test this round, and whether or not the restriction that the samples be in distinct rows and columns is feasible.
    - The function first checks if the number of unclassified patients is less than or equal to the maximum number of tests inputted by the user. If so, we test all remaining samples.
    - If the testing matrix passes the first check, it then checks if the minimum of the number of rows and the number of columns which have not been classified is less than the maximum number of tests divided by 2. If so, it chooses to test the maximum number of tests inputted by the user without the restriction that they be in distinct rows and columns.
    - If the testing matrix passes both checks, the function then chooses to test the maximum number of tests inputted by the user with the restriction that all samples tested be in distinct rows and columns.
  - Next, the function tests the number of samples chosen.
    - The function extracts the indices of the sample corresponding to the highest sum of row and columns averages, stores the observable VL for that patient and subtracts the VL divided by n from the corresponding row and column averages.
    - It then zeros out the VL entry for that individual to reflect that we have tested that patient. Notice that for the 'y' matrix, we only zero out the true VL value. Y is the partially reduced matrix we will pass into the sensitivity function.
    - It increases the number of tests by 1 for each sample tested.
    - This loop repeats for each sample chosen for testing.
  - Increases the number of round by 1.
  - Checks if any row/column averages are less than *t/n*.
  - Checks if all samples in a row/column have been tested.
- Once the while loop completes, all patients are classified, and the function returns a dataframe made up of the partially reduced matrix y (for sensitvity calculations) with added columns for number of tests and rounds taken to classify all patients.

Below is the R code for the MSS method as described above, including the sums function.

```
# creates data frame of all possible sums of row and column averages
# indexed by the ith row and jth column
# if an element is zero, it stays zero regardless of the sum
# x is a testing matrix
sums <- function(x, matsize){
  s <- matrix(nrow = matsize, ncol = matsize)
  sums <- data.frame(s)
  for (i in 1:matsize){
    for (j in 1:matsize){
      if (x[i,j]==0){
        sums[i,j]=0}
      else{sums[i,j] <- x$rows[i] + x$cols[j]}}}
```

```
  return(sums)
}

# Function which classifies all patients using the MSS Method
# x and y are identical testing matrices created by one.matrix described abov
e
reduce.mat.smt <- function(x, y, matsize, cutoff, tstperd, lowlimit){
  # tracks number of tests
  t <- matsize*2
  # tracks number of rounds
  rounds <- 1
  # diagnostic; irrelevant
  check <- matrix(nrow=0, ncol=2)
  # Calculates prevaence metrics for diagnostics
  prev <- prev(x, matsize, cutoff)
  # function that checks every row and column average.  If less than t/n,
  # this function classifies all patients in that row/column as not experienc
ing
  # treatment failure and zeros out that row/column
  x <- zerocolrow(x,matsize,cutoff, lowlimit)

  # Start of the while loop which determines when to stop testing
  while (max(x$cols) > cutoff/matsize & max(x$rows) > cutoff/matsize){
    # tracks how many rows and columns can contain failures
    num.rows.fail <- length(x$rows[x$rows > cutoff/matsize])
    num.cols.fail <- length(x$cols[x$cols > cutoff/matsize])
    # calculates the minimum of the above two numbers
    # useful in choosing how many tests to conduct
    min.num.fails <- min(num.rows.fail, num.cols.fail)
    # function that creates data frame of all possible sums of row and column
totals
    # indexed by the ith row and jth column corresponding to the ith, jth sam
ple.
    # If a patient has been classified, their sum will be zero regardless of
their
    # corresponding row and column averages
    sums <- sums(x,matsize)
    z <- NULL
    d <- matrix(nrow=0, ncol=2)
    # orders the sums in a list in descending order
    z <- order(sums, decreasing=TRUE)
    # extracts the indicies corresponding to each sum total
    # only extracts indices of patients not yet classified
    for (k in 1:length(z)){
      j <- ceiling(z[k]/matsize)
      if (z[k] %% matsize == 0){i <- matsize}
      else{i <- z[k] %% matsize}
      if (x[i,j] == 0){}
      else{d <- rbind(d, c(i,j))}
    }
```

```
    # d is a dataframe of the indices of all unclassified patients from highe
st sum
    # to lowest
    d <- data.frame(d)
    # the number of unclassfied patients remaining
    max.num.fails <- length(d[,1])

    # The next three if/else statements choose the number of tests to conduct
this
    # round

    # This determines if the number of unclassified patients is less than
    # or equal to our maximum number of tests per round
    if (max.num.fails <= tstperd){
      for (i in 1:length(d[,1])){
        # extracts the indices from data frame d
        a <- d[i,1]
        b <- d[i,2]
        # extracts the tested value (includes measurement error)
        xerror <- x[[a, (b+matsize)]]/matsize
        # Subtracts the appropriate value for the corresponding row and colum
n
        x$rows[a] <- x$rows[a] - xerror
        x$cols[b] <- x$cols[b] - xerror
        # zeros out the true value and tested value from the x matrix.
        # This allows us to track which patients have been classified.
        x[[a,(matsize+b)]] = 0
        x[[a,b]] = 0
        # zeros out the true value in the y matrix to indicate that we tested
        # this patient.  Useful for calculating sensitivity in a later functi
on.
        y[[a,b]] = 0
        # increases our number of tests by 1.
        t <- t+1
        # re-declares d as a data frame.  Important for when d has only 1 row
.
        d <- data.frame(d)
      }
    }
    # If we pass the above check, this determines if the minimum of the numbe
r of
    # rows/columns which could contain a failure is less than the maximum num
ber of
    # tests divided by 2.  If so, we relax the restriction that all patients
    # tested be in distinct rows and columns.
    else if (min.num.fails < tstperd/2){
      for (i in 1:tstperd){
        # extracts the indices from data frame d
        a <- d[i,1]
        b <- d[i,2]
```

```
        # extracts the tested value (includes measurement error)
        xerror <- x[[a, (b+matsize)]]/matsize
        # Subtracts the appropriate value for the corresponding row and colum
n
        x$rows[a] <- x$rows[a] - xerror
        x$cols[b] <- x$cols[b] - xerror
        # zeros out the true value and tested value from the x matrix.
        # This allows us to track which patients have been classified.
        x[[a,(matsize+b)]] = 0
        x[[a,b]] = 0
        # zeros out the true value in the y matrix to indicate that we tested
        # this patient.  Useful for calculating sensitivity in a later functi
on.
        y[[a,b]] = 0
        # increases our number of tests by 1.
        t <- t+1
      }
    }
    # If we pass the above two checks, we will test the maximum number of tes
ts
    # ensuring that the samples tested are in distinct rows and columns, star
ting
    # with the highest sum
    else{
      for (i in 1:tstperd){
        if (length(d[,1]) == 0){}
        else{
          # extracts the indices from data frame d
          a <- d[1,1]
          b <- d[1,2]
          # extracts the tested value (includes measurement error)
          xerror <- x[[a, (b+matsize)]]/matsize
          # Subtracts the appropriate value for the corresponding row and col
umn
          x$rows[a] <- x$rows[a] - xerror
          x$cols[b] <- x$cols[b] - xerror
          # zeros out the true value and tested value from the x matrix.
          # This allows us to track which patients have been classified.
          x[[a,(matsize+b)]] = 0
          x[[a,b]] = 0
          # zeros out the true value in the y matrix to indicate that we test
ed
          # this patient.  Useful for calculating sensitivity in a later func
tion.
          y[[a,b]] = 0
          # This removes all sample indices in the same row or column as the
          # sample neing tested.
          d <- data.frame(d[d[,1] != a & d[,2] != b,])
          # increases our number of tests by 1.
          t <- t+1
```

```
        }
      }
    }
    # Increases our number of testing rounds by 1
    rounds = rounds+1
    # function that checks every row and column average.  If less than t/n,
    # this function classifies all patients in that row/column as not experie
ncing
    # treatment failure and zeros out that row/column
    x <- zerocolrow(x,matsize,cutoff,lowlimit)
    # Function that checks if we have classified all patients in a row/column
    # and if so, zeros out that row/column
    x <- checkrowcol(x,matsize)
  }
  # this returns matrix y with the true values zeroed out for every patient
  # we actually tested (not just classified).  A later function will compare
these to
  # the observable values to calculate sensitivity.  It also attachs extra co
lumns onto
  # matrix y as a way of also returning number of tests,rounds and prev metri
cs
  return (data.frame(cbind(y,t, rounds, prev[1], prev[2], prev[3], prev[4],
                           prev[5], prev[6], prev[7], prev[8], prev[9], prev[
10],
                           prev[11], prev[12])))
}
```

A3.2.2 *Minipool + Algorithm (Mini)*

Below are steps for classifying a matrix of samples using the Mini method; reduce.mat.mini.

- Tracks the number of tests and rounds. The number of tests begins at our matrix size, because the Mini method does not pool across rows and columns. Number of rounds begins at 1.
- Checks if any row (a minipool) is less than *t/n*, and if so, classifies those patients as non-treatment failures.
- Start of the 'while' loop whih determines when all patients are classified, i.e., all row averages are less than *t/n*.
  - Finds each row that hasn't been classified, and tests one sample in each of those rows, starting with the first, unclassified sample in those rows. Since the samples were placed in the testing matrix randomly, this is equivalent to randomly selecting a sample in each given row.
  - Stores the VL for each patient tested and appropriately subtracts their value divided by n from the corresponding row average.
  - Increases the number of tests by 1 for each sample tested.
  - Increases the number of rounds by 1
  - checks if any row average is now less than *t/n*.

  - Checks if we have tested all samples in a given row, and if so zeroes out that row average.
- Once the 'while' loop completes, and all patients are classified, returns matrix 'y' for sensitivty calculation as well as number of tests, rounds, and prevalence metrics.

Below is the code for classifying alll patients using the Mini method. Included is the reduce.mat function reduce.mat.mini as well as the functions unique to the minipool methods.

```
# Checks if any row average is less than t/n, andif so, zeroes out that avera
ge
# as well as all elemtns in that row
zero.row.mini <- function(x, matsize, cutoff){
  for (i in 1:matsize){
    if (x$rows[i] < cutoff/matsize){
      x$rows[i] = 0
      for (j in 1:matsize){
        x[i,j] = 0
      }
    }
  }
  return(x)
}

# Checks if all samples in a given row have been classified
# If so, it zeroes out that row average
check.row.mini <- function(x, matsize){
  for (i in 1:matsize){
    if(sum(x[i,1:matsize]) == 0){x$rows[i] = 0}
    else{}
  }
  return(x)
}

# Function that classifies all patients using the Mini Method
# pop is a vecotr of VL's
reduce.mat.mini <- function(pop, x, y, matsize, cutoff){
  # tracks number of tests
  t <- matsize
  # tracks number of rounds
  rounds <- 1
  # tracks prevalence metrics
  prev <- prev(x, matsize, cutoff)
  # zeroes out any row with average less than t/n
  x <- zero.row.mini(x, matsize, cutoff)
  # start of while loop which finishes when all patient are classified
  while (max(x$rows) > (cutoff/matsize)){
    # Initializing a vector
    xerror <- NULL
    # loops over all rows, only doing thing with unclassified rows
    for (i in 1:matsize){
```

```
      if (x$rows[i] == 0){}
      else{
        index <- 1
        # searches for the first uclassified sample in a row
        while (x[i, index] == 0){
          index <- index + 1
        }
        # stores the observable VL for the first unclassified sample
        # in this row, and adjusts the row average accordingly
        xerror <- x[[i,(index+matsize)]]/matsize
        x$rows[i] <- x$rows[i] - xerror
        # zeroes out that sample to signify it is classified
        x[[i,(index+matsize)]] <- 0
        x[i, index] = 0
        # zeroes out the sample to signify  we tested the sample
        y[i, index] = 0
        # increases number of tests by 1
        t <- t+1
      }
    }
    # increases number of rounds by 1
    rounds <- rounds+1
    # zeroes out any row with average less than t/n
    x <- zero.row.mini(x, matsize, cutoff)
    # zeroes out row averages if we have classified all patients in a row
    x <- check.row.mini(x, matsize)
  }
  # returns a partially reduced matrix 'y' as well as number of tests, rounds
and
  # failure prevalence metrics
  return (data.frame(cbind(y,t, rounds, prev[1], prev[2], prev[3], prev[4],
                           prev[5], prev[6], prev[7], prev[8], prev[9], prev[
10],
                           prev[11], prev[12])))
}
```

A3.2.3 *Linear Regression (Linreg)*

Below are steps for classifying a matrix of samples using the Linreg method; reduce.mat.linreg.

- Tracks number of tests and rounds; initializing at twice the matrix size and 1, respectively.
- Tracks failure prevalence metrics.
- Zeroes out rows/column whose average is less than t/n.
- Tracks fitting-precision metrics. Fitting-precision metrics refers to how close we are able to match the linear regression estimates to the observed row and column averages.
- Start of the 'while' loop which determines when all patients have been classified.
  - Creates an soe.matrix from our testing matrix. Due to measurement error, the sum of the row averages does not necessarily equal the sum of the column averages. So we take the mean of the sum of row averages and the sum of column averages, and

proportionally alter the individual row and column averages to sum to that mean. This creates a testing matrix which has a solution. Code for this function is also included below.

  - Calculates the number of tests to be conducted this round. This is chosen to be the minimum of; the number of tests inputted by the user, the number of unclassified rows and the number of unclassified columns.
  - Creates a matrix solution which most closely agrees with our linear regression estimates of viral load using the function mat.linreg2 (this function code is also below). A new solution is created each testing round, incorporating the information from the indvidual tests.
  - Extracts the indices of the highest VL's as per our solution, and tests those samples; adjusting the row and column averages accordingly.
  - Zeroes out any row or columns whose average is less than t/n.
  - Zeroes out any row/column if all samples in that row/column have been tested.
  - Computes the maximum fitting-precision. Our output will include ths maximum which tells us how far off the worst fitting solution to our matrix is.

- Once the 'while' loop completes, all patients are classified, and it returns matrix 'y' for sensitivty calculation as well as number of tests, rounds, and prevalence and fitting-precision metrics.

Below is the code for classifying alll patients using the Linreg method. Included is the reduce.mat function, reduce.mat.linreg, as well as necessary functions not yet shown.

```
# This function takes a one.matrix and adjusts
# the row and column averages so that the sum of rows = sum of cols
soe.matrix <- function(x, matsize){
  tot <- (sum(x$rows) + sum(x$cols))/2
  rowdif <- (tot - sum(x$rows))/sum(x$rows)
  coldif <- (tot - sum(x$cols))/sum(x$cols)
  # proportionally alters the row and column averages
  x$rows <- x$rows*(1+rowdif)
  x$cols <- x$cols*(1+coldif)
  return(x)
}

# derives a matrix solution which matches our linear regression estimates
# to the observed row and column averages
# pop is the vector of VL's by themselves
# data is a data frame with VL and covariate info
# prec and precrd dictate how close we try to fit the regression estimates to
the row
# and column averages
# the b_start's are the prediction coefficients which can be inputted as anyt
hing
# they do not have to match the b_'s that generated the data.
# mat is a testing matrix
mat.linreg2 <- function(pop, data, mat, matsize, prec, precrd, b0star, b1star
, b2star, b3star){
```

```
  guess <- matrix(nrow=matsize, ncol=matsize)
  rows <- NULL
  cols <- NULL
  # Creates three matrices which will include each samples' row percent, colu
mn percent and
  # total percent.  The row percent is an individual sample's percent conrtib
ution to its row
  # total, based on regression estimates.  Column and total percent are compu
ted similarly.
  rowpercent <- matrix(nrow=matsize, ncol=matsize)
  colpercent <- matrix(nrow=matsize, ncol=matsize)
  totpercent <- matrix(nrow=matsize, ncol=matsize)
  # estimates each individual's viral load based on the prediction model
  for(i in 1:matsize){
    for (j in 1:matsize){
      # ensures we don't estimate a patient's VL if they've been classified
      if (mat[i,j]==0){guess[i,j]=0}
      else{
        # The 'match' call ensures we are matching the VL from pop to the ful
l patient
        # data in data frame 'data'.
        guess[i,j] <- 10^(b0star + b1star*data[match(mat[i,j], data$pop), 2]
+
                            b2star*data[match(mat[i,j], data$pop), 3] +
                            b3star*data[match(mat[i,j], data$pop), 2]*data[ma
tch(mat[i,j], data$pop), 3])
      }}}
  # Calculates all row and column totals as per our estimates
  for (i in 1:matsize){
    rows[i] <- sum(guess[i,])
    cols[i] <- sum(guess[,i])
  }
  # Calculates each individual's row, column and total percent contribution t
o the estimates
  for (i in 1:matsize){
    for (j in 1:matsize){
      rowpercent[i,j] <- guess[i,j]/rows[i]
      colpercent[i,j] <- guess[i,j]/cols[j]
      totpercent[i,j] <- guess[i,j]/sum(guess[,1:matsize])
    }
  }
  # multiplies the row, column and total percent by the actual observed row,
column and total
  # sums from our testing matrix and assigns the average of these 3 values to
our estimated
  # solution
  for (i in 1:matsize){
    for (j in 1:matsize){
      if (guess[i,j]==0){guess[i,j]=0}
      else{
```

```
        guess[i,j] <- (rowpercent[i,j]*matsize*mat$rows[i] + colpercent[i,j]*
matsize*mat$cols[j]
                       + totpercent[i,j]*matsize*sum(mat$rows))/3
      }}}
  # Calculated estimated row and column averages
  for (i in 1:matsize){
    rows[i] <- sum(guess[i,])/matsize
    cols[i] <- sum(guess[,i])/matsize
  }
  counter <- 1
  # while loop which chooses when to stop testing.  This occurs when
  # the maximum difference between the row and column averages and the estima
ted row and column
  # averages are less than prec.  It also iterates a maximum of precrd times.
  while (counter <= precrd & (max(abs(rows - mat$rows)) > prec | max(abs(cols
- mat$cols)) > prec)){
    # alters each individual estimate by a factor of
    # (observed column average/estimated column average) to proportionally ad
just
    # each estimate.
    for(i in 1:matsize){
      for(j in 1:matsize){
        if (guess[i,j]==0){guess[i,j]=0}
        else{
          guess[i,j] <- guess[i,j]*(mat$cols[j]/cols[j])}}}

    # re-calculates the estimated row averages
    for (i in 1:matsize){
      rows[i] <- sum(guess[i,])/matsize
      cols[i] <- sum(guess[,i])/matsize
    }
    # This chunk repreats the above steps but for the rows
    for(i in 1:matsize){
      for(j in 1:matsize){
        if (guess[i,j]==0){guess[i,j]=0}
        else{
          guess[i,j] <- guess[i,j]*(mat$rows[i]/rows[i])}}}

    for (i in 1:matsize){
      rows[i] <- sum(guess[i,])/matsize
      cols[i] <- sum(guess[,i])/matsize
    }
    counter <- counter+1
  }
  # tracks the maximum difference between the observed and estimate row and c
olumn averages
  prec.max <- max(max(abs(rows - mat$rows)), max(abs(cols - mat$cols)))
  # reutrns a matrix solution which matches the estimated and observed row an
d columns averages
  return(data.frame(cbind(guess, prec.max)))
```

```
}

# Function that classifies all patients using the Linreg Method
# pop is a vector of VL's coinciding with data
# data is full patient data with covariate information
# prec and precrd are for fitting precision described above
reduce.mat.linreg <- function(pop, data, x, y, matsize, prec, precrd,
                              b0star, b1star, b2star, b3star, cutoff, tstperd
, lowlimit){
  # tracks number of trials
  t <- matsize*2
  # tracks number of rounds
  rounds <- 1
  # tracks prevalence metrics
  prev <- prev(x, matsize, cutoff)
  # zeroes out rows and columns whose average is less than t/n.
  x <- zerocolrow(x,matsize,cutoff, lowlimit)
  prec.max <- NULL
  # Counter to track the max precision metric
  counter <- 1
  while (max(x$cols) > cutoff/matsize & max(x$rows) > cutoff/matsize){
    # alters original matrix rows and cols for the soe method
    z <- soe.matrix(x)
    xerror = NULL
    # calculates number of tests for this round.
    tstperd <- min(tstperd, min(length(x$rows[x$rows>0]), length(x$cols[x$col
s>0])))
    # creates a solution matching both regression estimates and observed
    # row and column averages
    guess <- mat.linreg2(pop, data, z, matsize, prec, precrd, b0star, b1star,
b2star, b3star)
    # tracks max precision
    prec.max[counter] <- guess[[1, (matsize+1)]]
    # estimated matrix solution
    guess <- data.matrix(guess[, 1:matsize])
    counter <- counter + 1
    d <- matrix(nrow=0, ncol=2)
    # extracts the indices for the highest estimated Sample
    # and zeroes out that element in our estimated solution
    for (i in 1:tstperd){
      num <- which.max(guess)
      if (num %% 10 == 0){k <- 10}
      else {k <- num %% 10}
      j <- ceiling(num/10)
      d <- rbind(d, c(k,j))
      guess[k,j] = 0
    }
    # tests the samples with the highest predicted VL
    for (i in 1:length(d[,1])){
      xerror[i] <- (x[[d[i,1],(matsize+d[i,2])]])/matsize
```

```
      a <- d[i,1]
      x$rows[a] <- x$rows[a] - xerror[i]
      b <- d[i,2]
      x$cols[b] <- x$cols[b] - xerror[i]
      x[[a,(matsize+b)]] = 0
      x[[a,b]] = 0
      y[[a,b]] = 0}
    # Updates number of rounds
    rounds = rounds+1
    # zeroes out rows and columns whose average is less than t/n.
    x <- zerocolrow(x,matsize,cutoff, lowlimit)
    # zeroes out any row/column where we've tested all samples
    x <- checkrowcol(x,matsize)
    # updates number of tests
    t <- t + length(d[,1])}
    # calculates our max precision metric
    prec.maxmax <- max(prec.max)
  # reutrns data frame with results
  return (data.frame(cbind(y,t, rounds, prev[1], prev[2], prev[3], prev[4],
                           prev[5], prev[6], prev[7], prev[8], prev[9], prev[
10],
                           prev[11], prev[12], prec.maxmax)))
}
```

A3.2.4 *Linear Regression Systems of Equations (LRSOE)*

Below are steps for classifying a matrix of samples using the LRSOE method; reduce.mat.linregsoe.

- Tracks number of tests and rounds; initializing at twice the matrix size and 1, respectively.
- Tracks failure prevalence metrics.
- Zeroes out rows/column whose average is less than t/n.
- Tracks fitting-precision metrics. Fitting-precision metrics refers to how close we are able to match the linear regression estimates to the observed row and column averages.
- Start of the 'while' loop which determines when all patients have been classified.
  - Creates an soe.matrix as in the Linreg method described above.
  - Creates a matrix solution which most closely agrees with our linear regression estimates of viral load using the function mat.linreg2 as in the Linreg method.
  - Increases the counter to properly track fitting-precision.
  - Converts our matrix solution VL's into weights
  - Uses function soe.solve.weights to find an LRSOE solution to the matrix, described in the steps below:
    - Beginning with the highest wieght, the function will assign the highest viral load value possible according to the row and column averages minus the lower detection limit multiplied by the number of unclassified samples which share a row or column with that sample.

    - All row and column averages are adjusted to reflect the newly-assigned values. These values cannot change again during the course of the solving the testing matrix.
    - Beginning with the next highest weight, the process repeats until we have a full solution to our testing matrix
  - Once we have our testing matrix solution, we test every sample our solution determines to be over the lower limit of detection. This process repeats every testing round with a new solution, so the number of tests varies per round of testing.
  - While testing, the function alters the row and column averages as in other methods, tracks the number of tests and rounds.
  - Zeroes out any row or columns whose average is less than t/n.
  - Zeroes out any row/column if all samples in that row/column have been tested.
  - Computes the maximum fitting-precision. Our output will include ths maximum which tells us how far off the worst fitting solution to our matrix is.

Below is the code for classifying all patients using the LRSOE method. Included is the reduce.mat function, reduce.mat.linregsoe, as well as the soe.solve.weights function described in the above steps.

```
# This function creates an LRSOE solution to the testing matrix by starting w
ith the highest
# individual weight and assigning the highest VL possible based on thr row av
erage, column
# average, and the lower limit of detection
# z is an soe testing matrix
# d is a matrix of row index, column index and estimated VL for each sample
soe.solve.weights <- function(z, d, matsize, lowlimit){
  # this counter tracks the indicies of the rows of d and this code will iter
ate over the top
  # rows of d until a solution to the matrix is found
  counter2 <- 1
  # fixed is a matrix which tracks the indices of samples already solved
  # this ensures no sample's estimated VL changes once fixed
  fixed <- matrix(nrow=0, ncol=2)
  # adds all zeroed out (fixed) samples to 'fixed' unless they're already the
re
  for (i in 1:matsize){
    for (j in 1:matsize){
      if (z[i,j] == 0){
        if (check.indices(fixed, c(i, j))){}
        else {fixed <- rbind(fixed, c(i,j))}
      }
      else {}
    }
  }

  while (length(fixed[,1]) < (matsize^2)){
    # tracks how many elements of z were 'zeroed' on a given interation
```

```
    num.boxes.zero <- 0

    # checks the first box to see if the row or col total for that box is gre
ater
    if ((z$rows[d[counter2,1]]) > (z$cols[d[counter2,2]])){

      # if the row is higher, we use the col total
      num <- z$cols[d[counter2,2]]
      # this applies the column total to the appropriate box and zeros out th
e other boxes
      for (i in 1:matsize){
        # This ensures we don't zero a box we've already fixed in our solutio
n
        if(check.indices(fixed, c(i, d[counter2,2]))){}
        else{
          # this zeros out all the boxes not fixed in the column
          z[i, d[counter2,2]] = lowlimit
          # tracks number of boxes zeroed
          num.boxes.zero <- num.boxes.zero + 1
          # alters each row total by our lower VL limit/matsize for each box
zeroed
          z$rows[i] <- z$rows[i] - lowlimit/matsize
          fixed <- rbind(fixed, c(i,  d[counter2,2]))}
      }

      box.num <- (num*matsize) - (lowlimit*(num.boxes.zero - 1))
      # applies our solution VL number to the appropriate box
      z[[d[counter2,1], d[counter2,2]]]= box.num
      # also zeros out the corresponding 'error values' (Probably not relevan
t)
      z[[d[counter2,1],(matsize+d[counter2,2])]] = 0
      # zeros out the column total
      z$cols[d[counter2,2]] <- 0
      # subtracts the num from the row total and adds back in lowlimit/count.
rowz that was
      # subtracted earlier
      z$rows[d[counter2,1]] <- z$rows[d[counter2,1]] - (box.num/matsize) + lo
wlimit/matsize
      # clears all coordinates in matrix d with the same col coordinate
      d <- subset(d, d[,2] != d[counter2,2])
      # reorders matrix d by weight
      d <- d[order(d[,3], decreasing = TRUE),]
      # ensures d is a matrix
      if (is.vector(d)){
        d <- matrix(d, nrow=1, ncol=3)}
      else{}
    }

    # the same comments apply to when column total > row total
```

```
    else{

      # extracts the row average
      num <- z$rows[d[counter2,1]]
      for (i in 1:matsize){
        # ensrues we do not touch boxes where we have fixed values of 0 or ot
herwise
        if(check.indices(fixed, c(d[counter2,1], i))){}
        else{
          # assigns fixed value zero to others in row
          z[d[counter2,1],i] = lowlimit
          # tracks number of boxes zeroed
          num.boxes.zero <- num.boxes.zero + 1
          # alters each row total by our lower VL limit/matsize for each box
zeroed
          z$cols[i] <- z$cols[i] - lowlimit/matsize
          fixed <- rbind(fixed, c(d[counter2,1], i))}
      }
      box.num <- (num*matsize) - (lowlimit*(num.boxes.zero - 1))
      z[[d[counter2,1], d[counter2,2]]]= box.num
      z[[d[counter2,1],(matsize+d[counter2,2])]] = 0
      z$rows[d[counter2,1]] <- 0
      z$cols[d[counter2,2]] <- z$cols[d[counter2,2]] - (box.num/matsize) + lo
wlimit/matsize
      d <- subset(d, d[,1] != d[counter2,1])
      d <- d[order(d[,3], decreasing = TRUE),]
      if (is.vector(d)){
        d <- matrix(d, nrow=1, ncol=3)}
      else{}
    }
  }
  # reutrns an LRSOE solution to the testing matrix
  return(z)
}

# This function classifies all patients using the LRSOE method
reduce.mat.linregsoe <- function(pop, data, x, y, matsize, cutoff,
                                 prec, precrd, b0star, b1star, b2star, b3star
, lowlimit = 0){


  # tracks number of tests
  t <- matsize*2
  # tracks number of rounds
  rounds <- 1
  # tracks prevalence metrics
  prev <- prev(x, matsize, cutoff)
  # zeroes out rows and columns whose average is less than t/n.
  x <- zerocolrow(x,matsize,cutoff, lowlimit)
```

```
  prec.max <- NULL
  # Counter to track the max precision metric
  counter <- 1
  # start of while loop which finishes when all patient are classified
  while (max(x$cols) > cutoff/matsize & max(x$rows) > cutoff/matsize){
    # alters original matrix rows and cols for the soe method
    z <- soe.matrix(x)
    d <- matrix(nrow=0, ncol=3)
    xerror = NULL
    # creates a solution matching both regression estimates and observed
    # row and column averages
    guess <- mat.linreg2(pop, data, z, matsize, prec, precrd, b0star, b1star,
b2star, b3star)
    # tracks max precision
    prec.max[counter] <- guess[[1, (matsize+1)]]
    # estimated matrix solution
    guess <- data.matrix(guess[, 1:matsize])
    # this counter is for prec.max tracking
    counter <- counter + 1
    # sum of all guessed VL values
    total <- sum(guess)
    # this converts VL values to weights and tracks the data in matrix d
    for (i in 1:matsize){
      for (j in 1:matsize){
        guess[i,j] <- guess[i,j]/total
        d <- rbind(d, c(i,j,guess[i,j]))
      }
    }
    # this orders matrix d by highest weight first
    d <- d[order(d[,3], decreasing = TRUE),]
    # This code ensures that d is a matrix, even if only 1 row
    if (is.vector(d)){
      d <- matrix(d, nrow=1, ncol=3)}
    else{}
    # this solves our soe matrix based on the weights
    # the result is an LRSOE solution to the testing matrix
    z <- soe.solve.weights(z, d, matsize, lowlimit)

    # start of our code to test VL values
    # extracts only our solution matrix used to decide which samples to test
    testmat <- data.frame(z[,1:matsize])
    # tracks number of boxes > 0 in our solution (testable boxes)
    over.limit <- testmat[testmat>lowlimit]
    index <- NULL
    # loops over the number of tests this round where number of tests
    # is equal to number of samples in our solution greater than the lower li
mit
    for (k in 1:length(over.limit)){
      # extracts the list coordinates of the highest estimated sample and con
verts to matrix
```

```
      # indices
      index <- head(order(testmat, decreasing=TRUE))
      # this code converts the list coordinates into matrix coords
      j <- ceiling(index[1]/matsize)
      if (index[1] %% 10 == 0){i <- 10}
      else{i <- index[1] %% 10}
      # zeros our our testmat box to ensure we do not test twice
      testmat[i,j] <- 0
      # extracts the observable value for the coordinate and divides by the m
atsize
      xerror <- (x[[i,(matsize+j)]])/matsize
      x$rows[i] <- x$rows[i] - xerror
      x$cols[j] <- x$cols[j] - xerror
      x[[i,(matsize+j)]] = 0
      x[[i,j]] = 0
      y[[i,j]] = 0
    }
    # updates number of rounds of testing
    rounds = rounds+1
    # zeroes out rows and columns whose average is less than t/n
    x <- zerocolrow(x,matsize,cutoff, lowlimit)
    # zeroes out any row/column where we've tested all samples
    x <- checkrowcol(x,matsize)
    # updates number of tests
    t <- t + length(over.limit)}
  # calculates max precision metric
  prec.maxmax <- max(prec.max)
  # returns result data frame
  return (data.frame(cbind(y,t, rounds, prev[1], prev[2], prev[3], prev[4],
                           prev[5], prev[6], prev[7], prev[8], prev[9], prev[
10],
                           prev[11], prev[12], prec.maxmax)))
}
```

A3.2.5 *Minipool with Covariates (Mini/Cov)*

Below are steps for classifying a matrix of samples using the Mini/Cov method; reduce.mat.mini.cov.

- Tracks number of tests and round; beginning at the matrix size and 1, respectively. As in the Mini method, we start with a matrix size number of tests, because we only pool across rows.
- Tracks prevalence metrics
- Zeroes our all rows with an average less than t/n.
- Start of the 'while' loop which determines when every patient has been classified
  - We estimate all samples' VL using our prediction model and the function mat.linreg.mini2 (code provided below)
  - For each row whose average is above t/n, we test the sample with the highest predicted VL.

  - Those tests' observed VL is subtracted accordingly from the corresponding row averages.
  - Updates the number of tests and rounds conducted
  - Zeroes out rows whose average is now less than t/n, and rows which all samples were tested.
- Once every patient is classified, returns the result data frame as in other methods described above.

Below is the code for classifying all patients using the Mini/Cov method. Included is the reduce.mat function, reduce.mat.mini.cov, as well as necessary functions not yet shown.

Note: When estimating VLs using a minipool, it is unexessary to fit the estimated solutions to the row averages. We only use the estimated VLs for choosing which individuals in a given row to test first. Therefore, the estimates are used only for ordering; fitting them to the row averages is irrelevant.

```
# funtion which estimates each patient's VL according to our prediction model
(for minipools)
mat.linreg.mini2 <- function(pop, data, mat, matsize, b0star, b1star, b2star,
b3star){
  guess <- matrix(nrow=matsize, ncol=matsize)
  # Estimates each individual's VL
  # the 'match' call matches VL from pop with complete covariate data in 'dat
a'.
  for(i in 1:matsize){
    for (j in 1:matsize){
      if (mat[i,j]==0){guess[i,j]=0}
      else{
        guess[i,j] <- 10^(b0star + b1star*data[match(mat[i,j], data$pop), 2]
+
                            b2star*data[match(mat[i,j], data$pop), 3] +
                            b3star*data[match(mat[i,j], data$pop), 2]*data[ma
tch(mat[i,j], data$pop), 3])
      }}}
  # returns our estimated values
  return(guess)
}

# Function that classifies all patients using the Mini/Cov method
reduce.mat.mini.cov <- function(pop, data, x, y, matsize, cutoff,
                                b0star, b1star, b2star, b3star){
  # tracks number of tests and rounds
  t <- matsize
  rounds <- 1
  # tracks prevalence metrics for diagnostic purposes
  prev <- prev(x, matsize, cutoff)
  # zeroes out rows whose average is less than t/n
  x <- zero.row.mini(x, matsize, cutoff)
```

```
  while (max(x$rows) > (cutoff/matsize)){
    xerror <- NULL
    # creates a matrix of estimated VLs based on the prediction model
    guess <- mat.linreg.mini2(pop, data, x, matsize, b0star, b1star, b2star,
b3star)
    # tests the 1 highest-estimated sample in each unclassified row
    for (i in 1:matsize){
      if (x$rows[i] == 0){}
      else{
        index <- which.max(guess[i,])
        xerror <- x[[i,(index+matsize)]]/matsize
        x$rows[i] <- x$rows[i] - xerror
        x[[i,(index+matsize)]] <- 0
        x[i, index] = 0
        y[i, index] = 0
        # updates number of tests
        t <- t+1
      }
    }
    # updates number of rounds
    rounds <- rounds+1
    # zeroes out any row whose average is less than t/n
    x <- zero.row.mini(x, matsize, cutoff)
    # zeroes out any rows where all patients in the row are classified
    x <- check.row.mini(x, matsize)
  }
  # returns result data frame
  return (data.frame(cbind(y,t, rounds, prev[1], prev[2], prev[3], prev[4],
                           prev[5], prev[6], prev[7], prev[8], prev[9], prev[
10],
                           prev[11], prev[12])))
}
```

A3.3 *Output Functions and Simulation Code*

The previous sections provide the description and code necessary for generating our data and implementing these methods in order to classify a matrix of patients. This section provides the output functions and actual code used in our simulations. Researchers should be able to reproduce our results exactly by then end of this section.

This next function is an example of a test.one function. The test.one functions take in the same inputs as the reduce.mat functions with the addition of the 'SE' input which is the standard deviation of measurement error on the log10 scale. The function first creates a testing matrix through one.matrix, applies the method to reduce the testing matrix, calculates sensitivity and outputs the results in a data frame. I only present one example of a test.one function as they are almost identical, and easy to reproduce.

```
# Function which simulates 1 matrix and outputs the performance results
test.one.linregsoe <- function(pop, data, matsize, prec, precrd, b0star, b1st
ar, b2star, b3star,
```

```
                                cutoff, SE, lowlimit=0){

  # creates a testing matrix of random entries from the population
  x <- one.matrix(pop, matsize, SE)
  # 2 exact copies of the original matrix
  y <- x
  z <- x
  # returns a matrix with zero'ed out boxes which were tested as well as
  # number of tests, rounds, prevalence metrics, and precision metric if appl
icable
  y <- reduce.mat.linregsoe(pop, data, x, y, matsize, cutoff,
                            prec, precrd, b0star, b1star, b2star, b3star, low
limit)
  # Calculates sensitivity
  x <- sens(y, z, matsize, cutoff)
  # Extracts the appropriate numbers to present in results
  t <- y[[1, (matsize*2 +3)]]
  rounds <- y[[1, (matsize*2 +4)]]
  tprev500 <- y[[1, (matsize*2 +5)]]
  tprevfail <- y[[1, (matsize*2 +6)]]
  eprev500 <- y[[1, (matsize*2 +7)]]
  eprevfail <- y[[1, (matsize*2 +8)]]
  tq90 <- y[[1, (matsize*2 +9)]]
  tq95 <- y[[1, (matsize*2 +10)]]
  tq99 <- y[[1, (matsize*2 +11)]]
  tq1 <- y[[1, (matsize*2 +12)]]
  eq90 <- y[[1, (matsize*2 +13)]]
  eq95 <- y[[1, (matsize*2 +14)]]
  eq99 <- y[[1, (matsize*2 +15)]]
  eq1 <- y[[1, (matsize*2 +16)]]
  prec.max <- y[[1, (matsize*2 +17)]]
  # returns a data frame of results
  return(c(x,t, rounds, tprev500, tprevfail, eprev500, eprevfail, tq90, tq95,
tq99, tq1,
           eq90, eq95, eq99, eq1, prec.max))
}
```

This test.one function can be easily applied to any method. Next is the output function. This function allows the user to input the number of simulated matrices, complete patient data, the testing matrix size, fitting-precision parameters, a failure cutoff, the standard devitation of measurement error, number of tests per round, the lower limit of detection and the prediction coefficients. The function then returns mean and quantile estimates for each method performance criteria as well as failure prevalence metrics.

Note that identical seeds are set before each test.one function in order to ensure that each method is testing identical matrices.

```
# Final simulation function which returns mean estimates of method performanc
e
pool.alg.test.soe3 <- function(reps, data, matsize, prec, precrd,
```

```
                                b0star=0, b1star=0, b2star=0, b3star=0,
                                cutoff, SE, tstperd, lowlimit){
  # clerical detail, sometimes named tstperrd and sometimes tstperd
  tstperrd <- tstperd
  # vector of VL's taken from 'data'
  pop <- data[,1]
  # Initializing performance vectors (matrices)
  linreg <- NULL
  smart <- NULL
  linregsoe <- NULL
  mini.rand <- NULL
  mini.pool <- NULL

  for (i in 1:reps){
    # takes the necessary sample size from the population to be used for the
testing matrix
    pop1 <- sample(pop, (matsize^2), replace=FALSE)
    # set isentical seeds to ensure fair comparison between methods
    set.seed(1)
    # tests one matrix for each method
    linreg1 <- test.one.linreg(pop1, data, matsize, prec, precrd, b0star, b1s
tar,
                               b2star, b3star, cutoff, tstperd, SE, lowlimit)

    set.seed(1)
    smart1 <- test.one.smt(pop1, matsize, cutoff, SE, tstperrd, lowlimit)

    set.seed(1)
    linregsoe1 <- test.one.linregsoe(pop1, data, matsize, prec, precrd, b0sta
r, b1star, b2star,
                                     b3star,cutoff, SE, lowlimit)

    set.seed(1)
    mini.rand1 <- test.one.mini.cov(pop1, data, matsize, cutoff, SE, b0star,
b1star, b2star, b3star)

    set.seed(1)
    mini.pool1 <- test.one.mini(pop, matsize, cutoff, SE)

    # removes tested subjects from population
    pop <- pop[! pop %in% pop1]
    # keeps track of th results from each matrix for each method
    linreg <- cbind(linreg, linreg1)
    smart <- cbind(smart, smart1)
    linregsoe <- cbind(linregsoe, linregsoe1)
    mini.rand <- cbind(mini.rand, mini.rand1)
    mini.pool <- cbind(mini.pool, mini.pool1)

  }
```

```
  # outputs the number of matrices with at least 1 failure
  # sensitivity for matrices without failure is NA
  mats.true <- linreg[1,]
  mats.error <- linreg[2,]

  # returns all performance and diagnostic results for all methods in a asing
le data frame
  # a function called pool.alg.row.names is presented after this function whi
ch assigns the
  # proper row names to the output
  return(c(num.mat.fail.true <- length(mats.true[!is.na(mats.true)]),
           num.mat.fail.error <- length(mats.error[!is.na(mats.error)]),
           ### Prevalence
           tprev500.min <- min(linreg[5,], na.rm=TRUE),
           tprev500.low5 <- quantile(linreg[5,], probs = .05, na.rm=TRUE),
           tprev500.mean <- mean(linreg[5,], na.rm=TRUE),
           tprev500.median <- median(linreg[5,], na.rm=TRUE),
           tprev500.high5 <- quantile(linreg[5,], probs = .95, na.rm=TRUE),
           tprev500.max <- max(linreg[5,], na.rm=TRUE),

           tprevfail.min <- min(linreg[6,], na.rm=TRUE),
           tprevfail.low5 <- quantile(linreg[6,], probs = .05, na.rm=TRUE),
           tprevfail.mean <- mean(linreg[6,], na.rm=TRUE),
           tprevfail.median <- median(linreg[6,], na.rm=TRUE),
           tprevfail.high5 <- quantile(linreg[6,], probs = .95, na.rm=TRUE),
           tprevfail.max <- max(linreg[6,], na.rm=TRUE),

           eprev500.min <- min(linreg[7,], na.rm=TRUE),
           eprev500.low5 <- quantile(linreg[7,], probs = .05, na.rm=TRUE),
           eprev500.mean <- mean(linreg[7,], na.rm=TRUE),
           eprev500.median <- median(linreg[7,], na.rm=TRUE),
           eprev500.high5 <- quantile(linreg[7,], probs = .95, na.rm=TRUE),
           eprev500.max <- max(linreg[7,], na.rm=TRUE),

           eprevfail.min <- min(linreg[8,], na.rm=TRUE),
           eprevfail.low5 <- quantile(linreg[8,], probs = .05, na.rm=TRUE),
           eprevfail.mean <- mean(linreg[8,], na.rm=TRUE),
           eprevfail.median <- median(linreg[8,], na.rm=TRUE),
           eprevfail.high5 <- quantile(linreg[8,], probs = .95, na.rm=TRUE),
           eprevfail.max <- max(linreg[8,], na.rm=TRUE),

           tq90.min <- min(linreg[9,], na.rm=TRUE),
           tq90.low5 <- quantile(linreg[9,], probs = .05, na.rm=TRUE),
           tq90.mean <- mean(linreg[9,], na.rm=TRUE),
           tq90.median <- median(linreg[9,], na.rm=TRUE),
           tq90.high5 <- quantile(linreg[9,], probs = .95, na.rm=TRUE),
           tq90.max <- max(linreg[9,], na.rm=TRUE),
```

```
            tq95.min <- min(linreg[10,], na.rm=TRUE),
            tq95.low5 <- quantile(linreg[10,], probs = .05, na.rm=TRUE),
            tq95.mean <- mean(linreg[10,], na.rm=TRUE),
            tq95.median <- median(linreg[10,], na.rm=TRUE),
            tq95.high5 <- quantile(linreg[10,], probs = .95, na.rm=TRUE),
            tq95.max <- max(linreg[10,], na.rm=TRUE),

            tq99.min <- min(linreg[11,], na.rm=TRUE),
            tq99.low5 <- quantile(linreg[11,], probs = .05, na.rm=TRUE),
            tq99.mean <- mean(linreg[11,], na.rm=TRUE),
            tq99.median <- median(linreg[11,], na.rm=TRUE),
            tq99.high5 <- quantile(linreg[11,], probs = .95, na.rm=TRUE),
            tq99.max <- max(linreg[11,], na.rm=TRUE),

            tq1.min <- min(linreg[12,], na.rm=TRUE),
            tq1.low5 <- quantile(linreg[12,], probs = .05, na.rm=TRUE),
            tq1.mean <- mean(linreg[12,], na.rm=TRUE),
            tq1.median <- median(linreg[12,], na.rm=TRUE),
            tq1.high5 <- quantile(linreg[12,], probs = .95, na.rm=TRUE),
            tq1.max <- max(linreg[12,], na.rm=TRUE),

            eq90.min <- min(linreg[13,], na.rm=TRUE),
            eq90.low5 <- quantile(linreg[13,], probs = .05, na.rm=TRUE),
            eq90.mean <- mean(linreg[13,], na.rm=TRUE),
            eq90.median <- median(linreg[13,], na.rm=TRUE),
            eq90.high5 <- quantile(linreg[13,], probs = .95, na.rm=TRUE),
            eq90.max <- max(linreg[13,], na.rm=TRUE),

            eq95.min <- min(linreg[14,], na.rm=TRUE),
            eq95.low5 <- quantile(linreg[14,], probs = .05, na.rm=TRUE),
            eq95.mean <- mean(linreg[14,], na.rm=TRUE),
            eq95.median <- median(linreg[14,], na.rm=TRUE),
            eq95.high5 <- quantile(linreg[14,], probs = .95, na.rm=TRUE),
            eq95.max <- max(linreg[14,], na.rm=TRUE),

            eq99.min <- min(linreg[15,], na.rm=TRUE),
            eq99.low5 <- quantile(linreg[15,], probs = .05, na.rm=TRUE),
            eq99.mean <- mean(linreg[15,], na.rm=TRUE),
            eq99.median <- median(linreg[15,], na.rm=TRUE),
            eq99.high5 <- quantile(linreg[15,], probs = .95, na.rm=TRUE),
            eq99.max <- max(linreg[15,], na.rm=TRUE),

            eq1.min <- min(linreg[16,], na.rm=TRUE),
            eq1.low5 <- quantile(linreg[16,], probs = .05, na.rm=TRUE),
            eq1.mean <- mean(linreg[16,], na.rm=TRUE),
            eq1.median <- median(linreg[16,], na.rm=TRUE),
            eq1.high5 <- quantile(linreg[16,], probs = .95, na.rm=TRUE),
            eq1.max <- max(linreg[16,], na.rm=TRUE),
```

```
            ### Linreg method output
            true.sens.min.linreg <- min(linreg[1,], na.rm=TRUE),
            true.sens.low5.linreg <- quantile(linreg[1,], probs = .05, na.rm=T
RUE),
            true.sens.mean.linreg <- mean(linreg[1,], na.rm=TRUE),
            true.sens.median.linreg <- median(linreg[1,], na.rm=TRUE),
            true.sense.high5.linreg <- quantile(linreg[1,], probs = .95, na.rm
=TRUE),
            true.sens.max.linreg <- max(linreg[1,], na.rm=TRUE),
            ### error sense
            error.sens.min.linreg <- min(linreg[2,], na.rm=TRUE),
            error.sens.low5.linreg <- quantile(linreg[2,], probs = .05, na.rm=
TRUE),
            error.sens.mean.linreg <- mean(linreg[2,], na.rm=TRUE),
            error.sens.median.linreg <- median(linreg[2,], na.rm=TRUE),
            error.sense.high5.linreg <- quantile(linreg[2,], probs = .95, na.r
m=TRUE),
            error.sens.max.linreg <- max(linreg[2,], na.rm=TRUE),
            ### trials
            trials.min.linreg <- min(linreg[3,], na.rm=TRUE),
            trials.low5.linreg <- quantile(linreg[3,], probs = .05, na.rm=TRUE
),
            trials.mean.linreg <- mean(linreg[3,], na.rm=TRUE),
            trials.median.linreg <- median(linreg[3,], na.rm=TRUE),
            trials.high5.linreg <- quantile(linreg[3,], probs = .95, na.rm=TRU
E),
            trials.max.linreg <- max(linreg[3,], na.rm=TRUE),
            ### relative efficiency
            rel.eff.linreg <- 1 - (trials.mean.linreg/(matsize^2)),
            ### rounds
            rounds.min.linreg <- min(linreg[4,], na.rm=TRUE),
            rounds.low5.linreg <- quantile(linreg[4,], probs = .05, na.rm=TRUE
),
            rounds.mean.linreg <- mean(linreg[4,], na.rm=TRUE),
            rounds.median.linreg <- median(linreg[4,], na.rm=TRUE),
            rounds.high5.linreg <- quantile(linreg[4,], probs = .95, na.rm=TRU
E),
            rounds.max.linreg <- max(linreg[4,], na.rm=TRUE),
            #### max precision for linreg
            precmax.min.linreg <- min(linreg[17,], na.rm=TRUE),
            precmax.low5.linreg <- quantile(linreg[17,], probs = .05, na.rm=TR
UE),
            precmax.mean.linreg <- mean(linreg[17,], na.rm=TRUE),
            precmax.median.linreg <- median(linreg[17,], na.rm=TRUE),
            precmax.high5.linreg <- quantile(linreg[17,], probs = .95, na.rm=T
RUE),
            precmax.max.linreg <- max(linreg[17,], na.rm=TRUE),

            ### smart method output
```

```
            true.sens.min.smart <- min(smart[1,], na.rm=TRUE),
            true.sens.low5.smart <- quantile(smart[1,], probs = .05, na.rm=TRU
E),
            true.sens.mean.smart <- mean(smart[1,], na.rm=TRUE),
            true.sens.median.smart <- median(smart[1,], na.rm=TRUE),
            true.sense.high5.smart <- quantile(smart[1,], probs = .95, na.rm=T
RUE),
            true.sens.max.smart <- max(smart[1,], na.rm=TRUE),
            ### error sense
            error.sens.min.smart <- min(smart[2,], na.rm=TRUE),
            error.sens.low5.smart <- quantile(smart[2,], probs = .05, na.rm=TR
UE),
            error.sens.mean.smart <- mean(smart[2,], na.rm=TRUE),
            error.sens.median.smart <- median(smart[2,], na.rm=TRUE),
            error.sense.high5.smart <- quantile(smart[2,], probs = .95, na.rm=
TRUE),
            error.sens.max.smart <- max(smart[2,], na.rm=TRUE),
            ### trials
            trials.min.smart <- min(smart[3,], na.rm=TRUE),
            trials.low5.smart <- quantile(smart[3,], probs = .05, na.rm=TRUE),
            trials.mean.smart <- mean(smart[3,], na.rm=TRUE),
            trials.median.smart <- median(smart[3,], na.rm=TRUE),
            trials.high5.smart <- quantile(smart[3,], probs = .95, na.rm=TRUE)
,
            trials.max.smart <- max(smart[3,], na.rm=TRUE),
            ### relative efficiency
            rel.eff.smart <- 1 - (trials.mean.smart/(matsize^2)),
            ### rounds
            rounds.min.smart <- min(smart[4,], na.rm=TRUE),
            rounds.low5.smart <- quantile(smart[4,], probs = .05, na.rm=TRUE),
            rounds.mean.smart <- mean(smart[4,], na.rm=TRUE),
            rounds.median.smart <- median(smart[4,], na.rm=TRUE),
            rounds.high5.smart <- quantile(smart[4,], probs = .95, na.rm=TRUE)
,
            rounds.max.smart <- max(smart[4,], na.rm=TRUE),

            ### linregsoe method output
            true.sens.min.linregsoe <- min(linregsoe[1,], na.rm=TRUE),
            true.sens.low5.linregsoe <- quantile(linregsoe[1,], probs = .05, n
a.rm=TRUE),
            true.sens.mean.linregsoe <- mean(linregsoe[1,], na.rm=TRUE),
            true.sens.median.linregsoe <- median(linregsoe[1,], na.rm=TRUE),
            true.sense.high5.linregsoe <- quantile(linregsoe[1,], probs = .95,
na.rm=TRUE),
            true.sens.max.linregsoe <- max(linregsoe[1,], na.rm=TRUE),
            ### error sense
            error.sens.min.linregsoe <- min(linregsoe[2,], na.rm=TRUE),
            error.sens.low5.linregsoe <- quantile(linregsoe[2,], probs = .05,
na.rm=TRUE),
            error.sens.mean.linregsoe <- mean(linregsoe[2,], na.rm=TRUE),
```

```
            error.sens.median.linregsoe <- median(linregsoe[2,], na.rm=TRUE),
            error.sense.high5.linregsoe <- quantile(linregsoe[2,], probs = .95
, na.rm=TRUE),
            error.sens.max.linregsoe <- max(linregsoe[2,], na.rm=TRUE),
            ### trials
            trials.min.linregsoe <- min(linregsoe[3,], na.rm=TRUE),
            trials.low5.linregsoe <- quantile(linregsoe[3,], probs = .05, na.r
m=TRUE),
            trials.mean.linregsoe <- mean(linregsoe[3,], na.rm=TRUE),
            trials.median.linregsoe <- median(linregsoe[3,], na.rm=TRUE),
            trials.high5.linregsoe <- quantile(linregsoe[3,], probs = .95, na.
rm=TRUE),
            trials.max.linregsoe <- max(linregsoe[3,], na.rm=TRUE),
            ### relative efficiency
            rel.eff.linregsoe <- 1 - (trials.mean.linregsoe/(matsize^2)),
            ### rounds
            rounds.min.linregsoe <- min(linregsoe[4,], na.rm=TRUE),
            rounds.low5.linregsoe <- quantile(linregsoe[4,], probs = .05, na.r
m=TRUE),
            rounds.mean.linregsoe <- mean(linregsoe[4,], na.rm=TRUE),
            rounds.median.linregsoe <- median(linregsoe[4,], na.rm=TRUE),
            rounds.high5.linregsoe <- quantile(linregsoe[4,], probs = .95, na.
rm=TRUE),
            rounds.max.linregsoe <- max(linregsoe[4,], na.rm=TRUE),
            #### max precision for linregsoe
            precmax.min.linregsoe <- min(linregsoe[17,], na.rm=TRUE),
            precmax.low5.linregsoe <- quantile(linregsoe[17,], probs = .05, na
.rm=TRUE),
            precmax.mean.linregsoe <- mean(linregsoe[17,], na.rm=TRUE),
            precmax.median.linregsoe <- median(linregsoe[17,], na.rm=TRUE),
            precmax.high5.linregsoe <- quantile(linregsoe[17,], probs = .95, n
a.rm=TRUE),
            precmax.max.linregsoe <- max(linregsoe[17,], na.rm=TRUE),

            ### mini.rand method output
            true.sens.min.mini.rand <- min(mini.rand[1,], na.rm=TRUE),
            true.sens.low5.mini.rand <- quantile(mini.rand[1,], probs = .05, n
a.rm=TRUE),
            true.sens.mean.mini.rand <- mean(mini.rand[1,], na.rm=TRUE),
            true.sens.median.mini.rand <- median(mini.rand[1,], na.rm=TRUE),
            true.sense.high5.mini.rand <- quantile(mini.rand[1,], probs = .95,
na.rm=TRUE),
            true.sens.max.mini.rand <- max(mini.rand[1,], na.rm=TRUE),
            ### error sense
            error.sens.min.mini.rand <- min(mini.rand[2,], na.rm=TRUE),
            error.sens.low5.mini.rand <- quantile(mini.rand[2,], probs = .05,
na.rm=TRUE),
            error.sens.mean.mini.rand <- mean(mini.rand[2,], na.rm=TRUE),
            error.sens.median.mini.rand <- median(mini.rand[2,], na.rm=TRUE),
            error.sense.high5.mini.rand <- quantile(mini.rand[2,], probs = .95
```

```
, na.rm=TRUE),
            error.sens.max.mini.rand <- max(mini.rand[2,], na.rm=TRUE),
            ### trials
            trials.min.mini.rand <- min(mini.rand[3,], na.rm=TRUE),
            trials.low5.mini.rand <- quantile(mini.rand[3,], probs = .05, na.r
m=TRUE),
            trials.mean.mini.rand <- mean(mini.rand[3,], na.rm=TRUE),
            trials.median.mini.rand <- median(mini.rand[3,], na.rm=TRUE),
            trials.high5.mini.rand <- quantile(mini.rand[3,], probs = .95, na.
rm=TRUE),
            trials.max.mini.rand <- max(mini.rand[3,], na.rm=TRUE),
            ### relative efficiency
            rel.eff.mini.rand <- 1 - (trials.mean.mini.rand/(matsize^2)),
            ### rounds
            rounds.min.mini.rand <- min(mini.rand[4,], na.rm=TRUE),
            rounds.low5.mini.rand <- quantile(mini.rand[4,], probs = .05, na.r
m=TRUE),
            rounds.mean.mini.rand <- mean(mini.rand[4,], na.rm=TRUE),
            rounds.median.mini.rand <- median(mini.rand[4,], na.rm=TRUE),
            rounds.high5.mini.rand <- quantile(mini.rand[4,], probs = .95, na.
rm=TRUE),
            rounds.max.mini.rand <- max(mini.rand[4,], na.rm=TRUE),

            ### mini.pool method output
            true.sens.min.mini.pool <- min(mini.pool[1,], na.rm=TRUE),
            true.sens.low5.mini.pool <- quantile(mini.pool[1,], probs = .05, n
a.rm=TRUE),
            true.sens.mean.mini.pool <- mean(mini.pool[1,], na.rm=TRUE),
            true.sens.median.mini.pool <- median(mini.pool[1,], na.rm=TRUE),
            true.sense.high5.mini.pool <- quantile(mini.pool[1,], probs = .95,
na.rm=TRUE),
            true.sens.max.mini.pool <- max(mini.pool[1,], na.rm=TRUE),
            ### error sense
            error.sens.min.mini.pool <- min(mini.pool[2,], na.rm=TRUE),
            error.sens.low5.mini.pool <- quantile(mini.pool[2,], probs = .05,
na.rm=TRUE),
            error.sens.mean.mini.pool <- mean(mini.pool[2,], na.rm=TRUE),
            error.sens.median.mini.pool <- median(mini.pool[2,], na.rm=TRUE),
            error.sense.high5.mini.pool <- quantile(mini.pool[2,], probs = .95
, na.rm=TRUE),
            error.sens.max.mini.pool <- max(mini.pool[2,], na.rm=TRUE),
            ### trials
            trials.min.mini.pool <- min(mini.pool[3,], na.rm=TRUE),
            trials.low5.mini.pool <- quantile(mini.pool[3,], probs = .05, na.r
m=TRUE),
            trials.mean.mini.pool <- mean(mini.pool[3,], na.rm=TRUE),
            trials.median.mini.pool <- median(mini.pool[3,], na.rm=TRUE),
            trials.high5.mini.pool <- quantile(mini.pool[3,], probs = .95, na.
rm=TRUE),
            trials.max.mini.pool <- max(mini.pool[3,], na.rm=TRUE),
```

```
            ### relative efficiency
            rel.eff.mini.pool <- 1 - (trials.mean.mini.pool/(matsize^2)),
            ### rounds
            rounds.min.mini.pool <- min(mini.pool[4,], na.rm=TRUE),
            rounds.low5.mini.pool <- quantile(mini.pool[4,], probs = .05, na.r
m=TRUE),
            rounds.mean.mini.pool <- mean(mini.pool[4,], na.rm=TRUE),
            rounds.median.mini.pool <- median(mini.pool[4,], na.rm=TRUE),
            rounds.high5.mini.pool <- quantile(mini.pool[4,], probs = .95, na.
rm=TRUE),
            rounds.max.mini.pool <- max(mini.pool[4,], na.rm=TRUE)))
}

# This function attaches proper row names to the above output
# x is the result data frame
pool.alg.row.names <- function(x){
  row.names(x) <- c(

  "num.mat.fail.true",
  "num.mat.fail.error",

  "tprev500.min",
  "tprev500.low5",
  "tprev500.mean",
  "tprev500.median",
  "tprev500.high5",
  "tprev500.max",

  "tprevfail.min",
  "tprevfail.low5",
  "tprevfail.mean",
  "tprevfail.median",
  "tprevfail.high5",
  "tprevfail.max",

  "eprev500.min",
  "eprev500.low5",
  "eprev500.mean",
  "eprev500.median",
  "eprev500.high5",
  "eprev500.max",

  "eprevfail.min",
  "eprevfail.low5",
  "eprevfail.mean",
  "eprevfail.median",
  "eprevfail.high5",
  "eprevfail.max",
```

```
  "tq90.min",
  "tq90.low5",
  "tq90.mean",
  "tq90.median",
  "tq90.high5",
  "tq90.max",

  "tq95.min",
  "tq95.low5",
  "tq95.mean",
  "tq95.median",
  "tq95.high5",
  "tq95.max",

  "tq99.min",
  "tq99.low9",
  "tq99.mean",
  "tq99.median",
  "tq99.high9",
  "tq99.max",

  "tq1.min",
  "tq1.low9",
  "tq1.mean",
  "tq1.median",
  "tq1.high9",
  "tq1.max",

  "eq90.min",
  "eq90.low5",
  "eq90.mean",
  "eq90.median",
  "eq90.high5",
  "eq90.max",

  "eq95.min",
  "eq95.low5",
  "eq95.mean",
  "eq95.median",
  "eq95.high5",
  "eq95.max",

  "eq99.min",
  "eq99.low9",
  "eq99.mean",
  "eq99.median",
  "eq99.high9",
  "eq99.max",
```

```
  "eq1.min",
  "eq1.low9",
  "eq1.mean",
  "eq1.median",
  "eq1.high9",
  "eq1.max",

  "true.sens.min.linreg",
  "true.sens.low5.linreg",
  "true.sens.mean.linreg",
  "true.sens.median.linreg",
  "true.sense.high5.linreg",
  "true.sens.max.linreg",

  "error.sens.min.linreg",
  "error.sens.low5.linreg",
  "error.sens.mean.linreg",
  "error.sens.median.linreg",
  "error.sense.high5.linreg",
  "error.sens.max.linreg",

  "trials.min.linreg",
  "trials.low5.linreg",
  "trials.mean.linreg",
  "trials.median.linreg",
  "trials.high5.linreg",
  "trials.max.linreg",

  "rel.eff.linreg",

  "rounds.min.linreg",
  "rounds.low5.linreg",
  "rounds.mean.linreg",
  "rounds.median.linreg",
  "rounds.high5.linreg",
  "rounds.max.linreg",

  "precmax.min.linreg",
  "precmax.low5.linreg",
  "precmax.mean.linreg",
  "precmax.median.linreg",
  "precmax.high5.linreg",
  "precmax.max.linreg",

  "true.sens.min.smt",
  "true.sens.low5.smt",
  "true.sens.mean.smt",
  "true.sens.median.smt",
  "true.sense.high5.smt",
```

```
  "true.sens.max.smt",

  "error.sens.min.smt",
  "error.sens.low5.smt",
  "error.sens.mean.smt",
  "error.sens.median.smt",
  "error.sense.high5.smt",
  "error.sens.max.smt",

  "trials.min.smt",
  "trials.low5.smt",
  "trials.mean.smt",
  "trials.median.smt",
  "trials.high5.smt",
  "trials.max.smt",

  "rel.eff.smt",

  "rounds.min.smt",
  "rounds.low5.smt",
  "rounds.mean.smt",
  "rounds.median.smt",
  "rounds.high5.smt",
  "rounds.max.smt",


  "true.sens.min.linregsoe",
  "true.sens.low5.linregsoe",
  "true.sens.mean.linregsoe",
  "true.sens.median.linregsoe",
  "true.sense.high5.linregsoe",
  "true.sens.max.linregsoe",

  "error.sens.min.linregsoe",
  "error.sens.low5.linregsoe",
  "error.sens.mean.linregsoe",
  "error.sens.median.linregsoe",
  "error.sense.high5.linregsoe",
  "error.sens.max.linregsoe",

  "trials.min.linregsoe",
  "trials.low5.linregsoe",
  "trials.mean.linregsoe",
  "trials.median.linregsoe",
  "trials.high5.linregsoe",
  "trials.max.linregsoe",

  "rel.eff.linregsoe",
```

```
"rounds.min.linregsoe",
"rounds.low5.linregsoe",
"rounds.mean.linregsoe",
"rounds.median.linregsoe",
"rounds.high5.linregsoe",
"rounds.max.linregsoe",

"precmax.min.soelinregsoe",
"precmax.low5.linregsoe",
"precmax.mean.linregsoe",
"precmax.median.linregsoe",
"precmax.high5.linregsoe",
"precmax.max.linregsoe",

"true.sens.min.mini.rand",
"true.sens.low5.mini.rand",
"true.sens.mean.mini.rand",
"true.sens.median.mini.rand",
"true.sense.high5.mini.rand",
"true.sens.max.mini.rand",

"error.sens.min.mini.rand",
"error.sens.low5.mini.rand",
"error.sens.mean.mini.rand",
"error.sens.median.mini.rand",
"error.sense.high5.mini.rand",
"error.sens.max.mini.rand",

"trials.min.mini.rand",
"trials.low5.mini.rand",
"trials.mean.mini.rand",
"trials.median.mini.rand",
"trials.high5.mini.rand",
"trials.max.mini.rand",

"rel.eff.mini.rand",

"rounds.min.mini.rand",
"rounds.low5.mini.rand",
"rounds.mean.mini.rand",
"rounds.median.mini.rand",
"rounds.high5.mini.rand",
"rounds.max.mini.rand",

"true.sens.min.mini.pool",
"true.sens.low5.mini.pool",
"true.sens.mean.mini.pool",
"true.sens.median.mini.pool",
"true.sense.high5.mini.pool",
```

```
  "true.sens.max.mini.pool",

  "error.sens.min.mini.pool",
  "error.sens.low5.mini.pool",
  "error.sens.mean.mini.pool",
  "error.sens.median.mini.pool",
  "error.sense.high5.mini.pool",
  "error.sens.max.mini.pool",

  "trials.min.mini.pool",
  "trials.low5.mini.pool",
  "trials.mean.mini.pool",
  "trials.median.mini.pool",
  "trials.high5.mini.pool",
  "trials.max.mini.pool",

  "rel.eff.mini.pool",

  "rounds.min.mini.pool",
  "rounds.low5.mini.pool",
  "rounds.mean.mini.pool",
  "rounds.median.mini.pool",
  "rounds.high5.mini.pool",
  "rounds.max.mini.pool")
  return(x)}
```

Below is the actual code we used to produce the simulation results presented in this paper.

```
data <- read.table("data2025.shapes.6.0.1.05betas1..05.1..05sd.5")

set.seed(18)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.05, b2star=1, b3star= 0.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1AGAIGprev20pf25.R")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.ignore.adh.prev20pf25.R")
```

```
data <- read.table("data1025.shapes.4.0.0.33betas1..05.1..05sd.5")


set.seed(16)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1AGAIGprev10pf25.R")


set.seed(16)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.ignore.adh.prev10pf25.R")


data <- read.table("data525.shapes.7.0.0.3betas1..05.1..05sd.5")


set.seed(14)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1AGAIGprev5pf25.R")

set.seed(14)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.ignore.adh.prev5pf25.R")


data <- read.table("data125.shapes.8.0.0.05betas1..05.1..05sd.5")
```

```
set.seed(12)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1AGAIGprev1pf25.R")

set.seed(12)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.ignore.adh.prev1pf25.R")

data <- read.table("data2025.shapes.6.0.1.05betas1..05.1..05sd.5")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.ignore.pf.prev20pf25.R")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-0.05, b2star=-1, b3star= -0.0
5,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.reverse.prev20pf25.R")


data <- read.table("data1025.shapes.4.0.0.33betas1..05.1..05sd.5")


set.seed(16)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)
```

```
result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.ignore.pf.prev10pf25.R")


set.seed(16)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -0.05
,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.reverse.prev10pf25.R")


data <- read.table("data525.shapes.7.0.0.3betas1..05.1..05sd.5")


set.seed(14)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.ignore.pf.prev5pf25.R")

set.seed(14)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.reverse.prev5pf25.R")

data <- read.table("data125.shapes.8.0.0.05betas1..05.1..05sd.5")


set.seed(12)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.ignore.pf.prev1pf25.R")
```

```
set.seed(12)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.reverse.prev1pf25.R")

data <- read.table("data2025perm40.shapes.6.0.1.05betas1..05.1..05sd.5")


set.seed(18)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.05, b2star=1, b3star= 0.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.AGAIGprev20pf25.R")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.ignore.adh.prev20pf25.R")



data <- read.table("data1025perm40.shapes.4.0.0.33betas1..05.1..05sd.5")


set.seed(16)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.AGAIGprev10pf25.R")

set.seed(16)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
```

```
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.ignore.adh.prev10pf25.R")


data <- read.table("data525perm40.shapes.7.0.0.3betas1..05.1..05sd.5")


set.seed(14)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.AGAIGprev5pf25.R")

set.seed(14)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.ignore.adh.prev5pf25.R")


data <- read.table("data125perm40.shapes.8.0.0.05betas1..05.1..05sd.5")


set.seed(12)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.05, b2star=1, b3star= .05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.AGAIGprev1pf25.R")

set.seed(12)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0, b2star=3, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
```

```
write.table(result2.data, file="adhere1.perm.ignore.adh.prev1pf25.R")

data <- read.table("data2025perm40.shapes.6.0.1.05betas1..05.1..05sd.5")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.ignore.pf.prev20pf25.R")

set.seed(18)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-0.05, b2star=-1, b3star= -0.0
5,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.reverse.prev20pf25.R")


data <- read.table("data1025perm40.shapes.4.0.0.33betas1..05.1..05sd.5")


set.seed(16)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=0.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.ignore.pf.prev10pf25.R")

set.seed(16)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -0.05
,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.reverse.prev10pf25.R")


data <- read.table("data525perm40.shapes.7.0.0.3betas1..05.1..05sd.5")
```

```
set.seed(14)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.ignore.pf.prev5pf25.R")

set.seed(14)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.reverse.prev5pf25.R")

data <- read.table("data125perm40.shapes.8.0.0.05betas1..05.1..05sd.5")


set.seed(12)
result1 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=1, b1star=.1, b2star=0, b3star= 0,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result1.data <- data.frame(result1)
result1.data <- pool.alg.row.names(result1.data)
write.table(result1.data, file="adhere1.perm.ignore.pf.prev1pf25.R")

set.seed(12)
result2 <- pool.alg.test.soe3(reps=500, data, matsize=10, prec=10, precrd=20,
                              b0star=6, b1star=-.05, b2star=-1, b3star= -.05,
                              cutoff=1000, SE=.12, tstperd=5, lowlimit=50)

result2.data <- data.frame(result2)
result2.data <- pool.alg.row.names(result2.data)
write.table(result2.data, file="adhere1.perm.reverse.prev1pf25.R")
```

This concludes the full description of our simulation methods. With this code, any researcher should be ale to reproduce our results exactly.